Finals Assignment 1 – Almendras

1. Differentiating Abstraction and Encapsulation

Many students confuse abstraction and encapsulation because both deal with hiding details, but they hide different kinds of details and for different reasons.

What they hide and why they hide it

Abstraction hides complex processes and only shows the essential idea of something. Its goal is to simplify how we interact with an object. For example, when using a “Printer” object, we only call .print() instead of worrying about ink levels, roller timing, or hardware processes. It hides complexity so the programmer focuses only on what the object does.

Encapsulation, on the other hand, hides data inside an object to protect it from unwanted access or modification. Its goal is security and maintaining valid state. For example, a BankAccount object hides its balance so outside code cannot directly set balance = -999. Instead, it forces controlled access through methods like deposit() or withdraw().

How they support each other

Abstraction and encapsulation work well together because encapsulation protects data internally, while abstraction provides a clean, simple interface externally. Abstraction tells the user what an object can do, while encapsulation controls how it does it and who can modify its data.

Examples where it matters

- ATM Machines: Users only see buttons (abstraction) while the actual money-handling logic is protected internally (encapsulation).

- Game Characters: The health value is hidden (encapsulation), while the player only uses simple actions like "attack" or "heal" (abstraction).

API Design: Libraries expose simple methods while hiding all internal algorithms.

2. Fake Sense of Security in Encapsulation

Encapsulation is often introduced as “data hiding,” but beginners get a false sense of security because simply making fields private does not automatically make data truly secure.

Why it creates a fake sense of security

Even with getters and setters, data can still be misused:

- A setter may allow invalid data (e.g., setAge(-5)).
- A getter might expose sensitive information.
- A setter might allow overwriting values freely.

So even though the variables are private, poorly written getters/setters can still break the object's integrity.

How to prevent or improve this

1. Add validation inside setters
- Example: prevent negative age or negative bank balance changes.
2. Avoid unnecessary setters
- If the value should never change, don't create a setter at all.
3. Use read-only objects or immutable classes
- Helps ensure the internal state stays safe.
4. Use defensive copying
- For lists or arrays, return copies instead of the original reference.
5. Limit access using access modifiers wisely
- private, protected, or package-private each serve different levels of protection.

Encapsulation is only as strong as the logic inside the methods protecting the data.

3. Abstraction Through Interfaces – When It Causes Issues

Interfaces require all methods to be implemented, but this can sometimes be a problem when a class cannot meaningfully implement some of the behaviors.

Imagine an interface:

```
interface Animal {
   void eat();
   void fly();
}
```

- A Bird class can implement both.

But a Dog class cannot fly, yet it is still forced to implement fly() because interfaces require full implementation.

This creates a situation where the interface does not match all possible objects.

There are two common solutions:

1. Split the interface (Interface Segregation Principle)

Break it into smaller, more specific interfaces:

```
interface Eater { void eat(); }
```

```
interface Flyer { void fly(); }
```

Now a Dog only implements Eater, not Flyer.

2. Use default methods with empty implementations (Java 8+)

```
interface Animal {
   void eat();
   default void fly() { /* not applicable */ }
}
```

This avoids forcing meaningless implementations, but must be used carefully.
The first solution (splitting) is usually considered better design.

References

- Gaddis, T. (2021). Starting out with Java: From control structures through objects (8th ed.). Pearson.
- Oracle. (2024). Trail: Learning the Java language – Classes and objects. https://docs.oracle.com/javase/tutorial/java/javaOO/
- Sebesta, R. W. (2018). Concepts of programming languages (12th ed.). Pearson.