Effective Encapsulation (Part I & II)

Part I ⭐⭐⭐

- Encapsulation is about Information Hiding.

Information Hiding

- Before, OOP, we had data & functions that operated on the data. The data was open—anyone could come in and operate on the data. Anytime the data was modified, the functions could have become invalid and would need to be modified/updated.
- **Main Purpose of Encapsulation: REDUCE COST OF CODE CHANGE.**
- Encapsulates logic or a piece of code around the certain functions that operate on it can contain the effect of the change rippling through the system. Modifying the data/data structure would affect only the code around it and not the entire system.
- This cost of change is much less today than it was 20-30 years ago, because of the modern IDEs and tools, etc.
    - However, if your library is used by others, then your changes will affect others' applications, which is still expensive.

Encapsulating a Field & Access Boundaries

- `public final` vs `private final` in `Java`:
  In case you intend to modify the implementation later on, use `private final`, else you can use `public final`.
    - Eg. If you do not want to reveal the data type of the variable — that is, you may not want to reveal that the variable is an `int`.
    - If you do not intend on modifying the implementation (eg. changing data type from `int` to `double`), then `private` or `public` is just ceremony — does not matter if it is `private final` or `public final`.

- **In Java (C++ or C#), encapsulation is at the class level.** You can't reach the private fields from outside the class, however, **an object of a class is able to access the private fields of another object of the same class**.
- In Ruby, by default, you can't get to the private field of another object from the first object. **Encapsulation in Ruby is at the instance level.**
- **In Scala, you can have Object Level, Class Level, Package Level encapsulation.**
    - By default, Scala has Class Level encapsulation. But, you can go to Package Level or Instance Level as well, depending on your needs.
    - **In** Scala, you can't access a private field from outside the class by default. Encapsulation in Scala is at the class level in this case.
    - However, if instead of `private var`, we write `private[this] var`, the other instance cannot access this field (= instance level encapsulation). In Scala, you can have Object Level, Class Level, or Package Level by changing `this` to a package name.
- Just putting `private` in front of a variable doesn't encapsulate it. (?)

It isn't about Security

- Encapsulation can easily be broken.
- Example of breaking encapsulation in Java:

```
Filed miles = Car.class.getDeclaredField("miles");
miles.setAccessible(true);
miles.set(car, 60);
```

- Example of breaking encapsulation in Ruby:

```
car.instance_eval do
    @miles = 60
end
```

- **Encapsulation isn’t about security. It is about making sure that the code is maintainable; That the change in a particular variable’s implementation doesn’t ripple through the code. It helps to preserve the invariant but not by ensuring that the malicious access won’t go through, but more so through accidental access.**

Real Purposes of Encapsulation

- Minimise cost of change.
  - So that the change to a variable (eg) does not ripple through the application and cause a havoc.
  - Especially in an application that is used by a third party. If you make a change, the third party will have to modify their code if you do not encapsulate.
- Preserve invariants.
  - To prevent accidental access.
  - Eg. if the distance needs to be positive, encapsulation can help preserve that invariant by preventing accidental negative values, however, if someone were to maliciously make that change, encapsulation cannot help.

Part II

Rethinking Encapsulation

- Don’t use the following techniques without having a real purpose, otherwise it’ll lead to a much more complex code than it should be. The degree to which encapsulation should be used depends on the problem, domain, application, the bounded complex, and usage scenario (much like abstraction).
- Only apply techniques where it makes sense. Don’t apply them just for the sake of applying.

Preserving Invariants

- **Invariant of an object is a never-changing property or a universal truth about that object** (⇒ constraint on an object that is true all the time).

    - E.g. if you have a car, which you drive, at any given time, the [total] distance travelled should not be less than the distance travelled so far.
    - Same with age → it only increases.
- Encapsulation helps to ensure the invariants are preserved and that objects transition from one valid state to another.

Minimising Cost of Change

- Not enough to make a field private. The *implementation* of the field must also be made private.
- Considering the example in the video, in Java, return `List` instead of `ArrayList` or `LinkedList`. `List` is an interface, while the others are implementation. Doing so, we aren't exposing how we are implementing the private fields. In case you want to change `ArrayList` to `LinkedList` or something else, the `main` class needn't change anything, as it is receiving a `List`.
- The implementation can be varied because only the interface is exposed.
- The implementation doesn't matter as long as the user of the abstraction gets the same performance.

Manage Concurrency

- Hide implementation details of private fields + synchronise the methods of the class so that the access is properly managed.

Safe Iteration

- A `for` loop fetches objects/elements one-by-one through a collection. `for each` is an iterator. Trying to add an element while using `for` loop will work fine. Trying to add an element while iterating using `for each` will lead to a problem. In Java → `ConcurrentModificationException`. Can't modify while iterating.