Day 2 OOPS Pillars || Abstraction || Encapsulation

Understanding OOPs of any language that's being used for the LLD is critical.

what is OOPs ?

Object Oriented Design (OOD) use 'Object-Oriented Methodology' to design a computational problem and its solution. To put it in simple terms - 'Basically allowing the writing of programs with the help of certain classes & real-time objects'.

[~ src : Medium art by AK Singh]

⭐ History of Programming :

1) Initially - Machine language (directly understood by CPU).
   ↳ Code written in Binary. eg: (0110010 ...)

   Drawbacks: Error prone, Not scalable, Too tedious ✗.

2) Next came - Assembly level languages :
   - Directly interact with hardware. Basically the hardware instruction set. eg: MOV A, 61H
   - NO error Mechanism, Machine dependent, Error prone
   - NO scalability

3) Upnext, 'Procedural Programming Language'

- Introduced functions, loops, Block statements
- eg: C - language.
- No good reusability, Complexity Maintainability.
  No real-world modelling.

(Basically seperation of data and code make it difficult).

So, our shift was towards OOPs

What is its Major advantage ? - 'Real-World Modelling'

The major idea is that - 'we should be able to model the applications that behave as real-world entities'.

(*) Benefits:

1) More intuitive and Natural Mapping (User, Car, Ride)
2) Secure encapsulation of Data - Control who can access & modify the data.
3) Code reusability using inheritance and interfaces.
4) Scalability through loosely coupled modules.

In Real world objects <=> objects
interact

→ Objects have i) characteristics (attributes)

ii) Behaviours (their functionality)

⊛ Modelling real-world entities in code :-

Objects: A real world 'thing' with attributes and behaviors.

Class : Blue Print (or) Template defining those attributes (fields) and behaviors (methods)

Instance: Actual existence of the object in a memory created from the class/blueprint.

⊛ Analogy : To better understand class, object, instance.

1) Imagine a 'Car' class as the blue print / design Template.

so, basically a car says : 'Every car should have a color, model and ability to drive'.

2) Object – The test car : The idea that a car exists. created from the blue print. It is still a thing.

3) Instance is like the actual car you bought having a number plate on.

⊛ Characteristics (attributes) Vs Behaviors (functionalities).

eg:  → Characteristics

- color
- Model
- wheels
- Doors
- engine ...

Behaviors (Methods)

- startEngine ( )
- stopCar ( )
- shiftGear ( );
- TurnLeft ( ); ...

Key Take away of OOPs from Procedural Prog. language:

with notion of classes: You have a blueprint.

Now - You can make as many cars as you want just by creating an object from the class.

eg: `Car *myCar1 = new Car ( );`

`Car *myCar2 = new Car ( );`

Without this, in Procedural Programming language, you will have declare all the characteristics and rewrite all methods for car 2. (Tedious)

To summarize, OOPS is the notion/idea of making objects interact with each other. (like in real world).

④ Pillars Of OOPs :-

1) Pillar 1 : ABSTRACTION → Definition :

'ABSTRACTION' – Hide unnecessary details from client. Expose only what's necessary for the functionality of the object.

In simple terms,

eg.1) You often use keyboard to type to the screen. Do you need to know how the key pressed is printed to the screen? NO

– As long as it does our job, we don't mind the inner functionality. You only care about pressing the key and the key-board doing it's job. The internal functionality is hidden from us. That key is an interface which allows us to interact with the computer.

eg 2)

- **Driving a Car**
  - **What you do**: Insert key, press pedals, turn steering wheel.
  - **What you don't need to know**: How the fuel-injection system works, how the transmission synchronizes gears, how the engine control unit computes ignition timing.
  - **Abstraction in action**: The car provides a simple interface ("start," "accelerate," "brake") and conceals all mechanical complexity under the hood.

Funny thing is that, we are using data abstraction all along the way beginning with writing if(), while(), but we don't mind how they are functioning internally.

```
/*

Abstract class -->
1. Act as an interface for the outside world to operate the car.
2. This abstract class tells 'WHAT' all it can do rather then 'HOW' it does that.
3. Since this is an abstract class we cannot directly create Objects of this class.
4. We need to Inherit it first and then that child class will have the responsibility to
provide implementation details of all the abstract (virtual) methods in the class.

5. In our real world example of Car, imagine you sitting in the car and able to operate
the car (startEngine, accelerate, brake, turn) just by pressing or moving some
pedals/buttons/ steer the wheel etc. You dont need to know how these things work, and
also they are hidden under the hood.
6. This Class 'Car' denotes that (pedals/buttons/steering wheel etc).

*/
class Car {
public:
    virtual void startEngine() = 0;
    virtual void shiftGear(int gear) = 0;
    virtual void accelerate() = 0;
    virtual void brake() = 0;
    virtual void stopEngine() = 0;
    virtual ~Car() {}
};
```

```
/*
1. This is a Concrete class (A class that provide implementation details of an abstract class).
Now anyone can make an Object of 'SportsCar' and can assign it to 'Car' (Parent class) pointer
(See main method for this)

2. In our real world example of Car, as you cannot have a real car by just having its body only
(all these buttons or steering wheel). You need to have the actual implementation of 'What' happens
when we press these buttons. 'SportsCar' class denotes that actual implementation.

3. Therefore, to denote a real world car in programming we created 2 classes.
One to denote all the user-interface like pedals, buttons, steering wheels etc ('Car' class).And,
Another one to denote the actual car with all the implementations of these buttons (SportsCar' class).

*/
```

⑧ Benefits of Abstraction :

1. Simplified interfaces – Clients focus on what an object does not how it does it.

2. Ease of maintenance – Internal changes.

eg: If you want to chang the engine, you can change it in the internal implementation. Clients won't have to do it.

3. Code Reusability – Multiple concrete classes can implement the same abstract interface

(eg: SportsCar, SUV, Electric Car ...)

4) Reduced Complexity: Large systems can be broken down and easily abstracted into modules.

---

⑨ PILLAR 2 : ENCAPSULATION

Def: Encapsulation – idea of Binding together an object's data (its state) and methods that operate on that data into a single unit, and controls access to its inner workings.

Analogy: Protecting internal state & group logic. Basically, imagine encapsulation to be a safe deposit box. Every thing

is safely put inside until you have the key to open it.

→ We make (classes) to bundle member (variables) and (methods) together.

```
Class Car {
    // Variables (// Characteristics)
    // Methods (// Behaviors)
}
```

Note: 

Encapsulation is also about 'Data Security'.

→ Basically a class might also have the data which needs to be kept secure from other objects.

How is Encapsulation different from Abstraction ?

- Encapsulation is about 'Data Security'.
- Abstraction is about 'Data Hiding'.

(★) Now, the question is How do you ensure 'Data Security ?'

Using ACCESS MODIFIERS in C++

1) Public : Members are accessible everywhere.
2) Private : Members are accessible only within the class.
3) Protected : Accessible in the class and the subclasses that inherited these classes.

(★) Getters and Setters with Validation:

Provides you control to modify the data, rather than exposing fields blindly.

```
/*
Encapsulation says 2 things:

1. An Object's Characteristics and its behaviour are encapsulated together within that Object.
2. All the characteristics or behaviors are not for everyone to access.
Object should provide data security.

We follow above 2 points about Object of real world in programming by:
1. Creating a class that act as a blueprint for Object creation. Class contain
all the characteristics (class variable) and behaviour (class methods) in one block,
encapsulating it together.
2. We introduce access modifiers (public, private, protected) etc to provide data
security to the class members.
3. And to access the private members of the object, we can use setters and getters
with control by putting validations or checks.

*/
class SportsCar {
private:
    string brand;
    string model;
    bool isEngineOn;
    int currentSpeed;
    int currentGear;

    //Introduce new variable to explain setters
    string tyreCompany;
```

⊛ Benefits of Encapsulation :-

1) **Prevents Tampering of data** : Prevents accidental / malicious misuse of internal instate.

2) **Maintainability** : Internal changes doesn't effect client's code.

3) **Clear Contracts** : Clients interact only via well-defined APIs (using public APIs)

4) **Modularity** : Code is well-organized into self-contained units, making testing easier & improve reusability.