📚 Lecture 02: OOPs Real-World Examples | OOPs Pillars | Abstraction | Encapsulation

---

🔷 Introduction - OOP Kyon Zaroori Hai?

- 🎯 **LLD ke liye Foundation:** OOP samajhna LLD (Low Level Design) ke liye strong foundation banata hai
- 🚀 Agar OOP clear ho gaya toh LLD samajhna bahut easy ho jata hai

---

🔶 Programming Languages ki Poori History

✅ 1. Machine Language (Sabse Pehli Generation)

- 💻 **Kya thi?** Binary codes (0s aur 1s) mein direct CPU se interact karti thi
- 📟 **Example Code:** `01010110101`
- ❌ **Problems:**
  - Bahut prone to errors - ek zero change karo, pura code change ho jata
  - Tedious process - aise code likhna bahut muskil
  - Bilkul bhi scalable nahi - large applications banana impossible

✅ 2. Assembly Language (Dusri Generation)

- 🔤 **Kya naya aaya?** English keywords/mnemonics ka use shuru
- 💾 **Example Code:** `MOV A, 61H`
- ❌ **Problems:**
  - Hardware ke saath tightly coupled - hardware change, code change
  - Phir bhi scalable nahi
  - Loops/methods properly implement nahi ho paate
  - Still tedious aur error-prone

✅ 3. Procedural Programming (Teesri Generation)

- 🏗️ **Kya naya aaya?** Functions, loops, blocks (if-else, switch) introduce kiye
- 📝 **Working Style:** "Do this, then do that" - recipe book ki tarah
- ✅ **Kya kar sakti thi?** Har wo cheez jo aaj hum use karte hain except OOP
- ❌ **Limitations:**
    - Complex problems solve karne mein inadequate
    - Large scale applications ke liye suitable nahi
    - Real-world modeling possible nahi tha

🧠 **Analogy:** Procedural programming ek recipe ki tarah hai - "pahle ye karo, phir wo karo". OOP real world ki tarah hai - "objects aapas mein baat karte hain"

🔷 OOP Kyon Aayi? - 3 Major Problems Solve Karne Ke Liye

📌 Problem 1: Real World Modeling Nahi Ho Pata Tha

- 🌍 **Real World Mein:** Har cheez object hai (car, mobile, human) aur objects aapas mein interact karte hain
- 💻 **Procedural Mein:** Sirf instructions ka sequence - real world jaisa feel nahi aata
- 🎯 **Solution:** OOP real world objects ko programming mein directly represent karta hai

📌 Problem 2: Data Security Nahi Thi

- 🔓 **Procedural Mein:** Koi bhi variable koi bhi change kar sakta tha
- 🚫 **Risk:** Important data easily corrupt ho sakta tha
- 🛡️ **Solution:** OOP mein data ko secure rakha ja sakta hai

📌 Problem 3: Scalability aur Reusability Nahi Thi

- 📈 **Large Applications:** Procedural mein complex applications banana mushkil
- 🔄 **Code Reuse:** Same code dobara use nahi kar paate the
- 🏗️ **Solution:** OOP scalable aur reusable code banane deta hai

---

🔶 Real World Objects - Poori Samajh

📌 Har Object Ke 2 Parts Hote Hain:

Characteristics (Properties/Data)

- Object ki unique identification
- **Example (Car ke liye):**
    - Engine 🚀
    - Brand 🏷️
    - Model 📝
    - Wheels 🛞

Behaviors (Methods/Functions)

- Object kya karta hai
- **Example (Car ke liye):**
    - Start karna 🚀
    - Stop karna 🛑
    - Gear shift karna ⚙️
    - Accelerate karna 🏎️
    - Apply Brakes 🚦

🧠 Detailed Car Example Table:

Characteristics	Behaviors
Brand (Ford) 🏷️	Start Engine 🚀
Model (Mustang) 📝	Stop Engine 🛑
Engine Status (On/Off) 🔧	Shift Gears ⚙️
Wheels (4) 🛞	Accelerate 🏎️
	Apply Brakes 🚦

🔷 OOP vs Procedural - Practical Example

❌ Procedural Approach Mein Problem:

```
string brand = "Ford";
string model = "Mustang";
bool isEngineOn = false;

void Drive(string carBrand, string carModel) {
    Start(carBrand, carModel);
    ShiftGear(carBrand, carModel, 1);
    Accelerate(carBrand, carModel);

}
```

Problems Procedural Mein:

- Har nayi car ke liye variables dobara declare karne padte
- Code complex aur maintain karna mushkil
- Real world jaisa interaction nahi dikhta
- Owner aur Car ka relation establish karna muskil

✅ OOP Approach Mein Solution:

cpp

```
class Car {
    string brand;
    string model;
public:
    void StartEngine() { /* implementation */ }
    void ShiftGear(int gear) { /* implementation */ }
};

class Owner {
    Car myCar;  // ✅ Real world jaisa!
public:
    void Drive() {
        myCar.StartEngine();  // ✅ Car Start ho jati
        myCar.ShiftGear(1);
    }

};
```

Advantages OOP Mein:

- Real world jaisa modeling
- Code clean aur readable
- Easily scalable
- Data security possible

💡 **Golden Tip:** OOP mein socho: "Kuch objects hain jo aapas mein interact kar rahe hain"

🔶 OOP Ke 4 Pillars

1. **Abstraction** 🤷‍♂️ - Implementation hide karna
2. **Encapsulation** 📦 - Data bundle aur secure karna
3. **Inheritance** 👨‍👩‍👧‍👦 - Code reuse
4. **Polymorphism** 🤷‍♂️ - Multiple forms

🔷 1. ABSTRACTION (Data Hiding) - Poori Detail

- **Definition:** "Unnecessary details hide, only necessary show"

- **Real Life Examples:**

🚗 **Car Drive Karna:**

- Aapko engine ka internal working janne ki need nahi
- Bas pedal dabana aana chahiye
- Hood khol kar engine dekh sakte hain, par zaroori nahi

📺 **TV Use Karna:**

- Remote ka use karna aata hai
- Internal wiring nahi janna padta
- TV kaise banata hai nahi janna padta

📱 **Mobile Use Karna:**

- Screen use karna aata hai
- Hardware kaise bana nahi janna padta

💡 **Technical Definition:** "Abstraction hides unnecessary details from a client and shows only what is necessary"

✅ Code Example: Abstract Car Class

cpp

```cpp
class Car {
public:
    // Abstract methods - sirf declaration, no implementation
    virtual void StartEngine() = 0;
    virtual void ShiftGear(int gear) = 0;
    virtual void Accelerate() = 0;
    virtual void ApplyBrake() = 0;
    virtual void StopEngine() = 0;
```

```cpp
    // Virtual destructor
    virtual ~Car() {}

};
```

✅ Implementation: SportsCar Class

cpp

```cpp
class SportsCar : public Car {

private:

    // Data members - characteristics

    string brand;

    string model;

    bool isEngineOn;

    int currentSpeed;

    int currentGear;



public:

    // Constructor

    SportsCar(string b, string m) : brand(b), model(m),

                                    isEngineOn(false),

                                    currentSpeed(0),

                                    currentGear(0) {}



    // Implementation of abstract methods

    void StartEngine() override {

        isEngineOn = true;

        cout << brand << " " << model << " engine started with a roar!" << endl;
```

```
}

void ShiftGear(int gear) override {
    if(!isEngineOn) {
        cout << "Engine is off! Cannot shift gear." << endl;
        return;
    }
    currentGear = gear;
    cout << "Shifted to gear " << gear << endl;
}

void Accelerate() override {
    if(!isEngineOn) {
        cout << "Start engine first!" << endl;
        return;
    }
    currentSpeed += 20;
    cout << "Accelerating to " << currentSpeed << " km/hr" << endl;
}

void ApplyBrake() override {
    currentSpeed -= 20;
    if(currentSpeed < 0) currentSpeed = 0;
    cout << "Braking... Current speed: " << currentSpeed << " km/hr" << endl;
```

```
    }

    void StopEngine() override {
        isEngineOn = false;
        currentSpeed = 0;
        currentGear = 0;
        cout << "Engine turned off" << endl;
    }
};
```

✅ Usage in Main Function:

cpp

```
int main() {
    // Car pointer pointing to SportsCar object
    Car* myCar = new SportsCar("Ford", "Mustang");

    // User ko implementation nahi janna - sirf interface use karna aana chahiye
    myCar->StartEngine();        // ✅ Abstraction in action
    myCar->ShiftGear(1);         // ✅ User ko nahi pata internal kaise work karta hai
    myCar->Accelerate();
    myCar->ShiftGear(2);
    myCar->Accelerate();
    myCar->ApplyBrake();
    myCar->StopEngine();
```

```
    delete myCar;

    return 0;

}
```

Output:

```
text

Ford Mustang engine started with a roar!

Shifted to gear 1

Accelerating to 20 km/hr

Shifted to gear 2

Accelerating to 40 km/hr

Braking... Current speed: 20 km/hr

Engine turned off
```

"Data aur behaviors ko ek unit (class) mein bundle karna + Data security provide karna"

🌟 Key Points – Abstraction

🔷 Definition:

Abstraction ka matlab hai **implementation details ko hide karke sirf essential features dikhana.**

✅ Main Points:

- User ko **internal implementation** janne ki zarurat nahi hoti.
- Sirf **interface (methods/functions)** ka use karna aana chahiye.
- Code **maintainable aur flexible** rehta hai.
- Real world ki tarah — hum har cheez ka internal working nahi jante (e.g. car drive karte hain but engine ka mechanism nahi jante).

💡 Example:

Programming Languages khud ek best example hain abstraction ka.
Aap `if`, `for`, `while` likhte ho —
par yeh kaise **machine code** mein convert hota hai, uski **details hide hoti hain.**

🔶 2. ENCAPSULATION (Data Security) - Poori Detail

📌 Encapsulation Kya Hai?

"Data aur behaviors ko ek unit (class) mein bundle karna + Data security provide karna"

🧠 Real Life Examples:

🚗 Car Ka Odometer:

- Directly change nahi kar sakte
- Driving se automatically badhta hai
- Agar directly change kar sakte toh gadi brand new dikh sakti thi!

🚗 Car Ki Current Speed:

- Directly set nahi kar sakte
- Accelerate karna padta hai gradually
- 0 se 100 directly set nahi kar sakte

💊 Medicine Capsule:

- Andar ki medicine protected rehti hai
- Outside se directly access nahi kar sakte

💡 Technical Definition:
"Encapsulation is the bundling of data and methods that operate on that data into a single unit (class), along with restricting direct access to some of the object's components"

✅ Code Example: Without Encapsulation (Problem)

cpp

```cpp
class SportsCar {
public:    // ❌ SAB KUCH PUBLIC - DANGEROUS!
    string brand;
    string model;
    bool isEngineOn;
    int currentSpeed;    // ❌ KOI BHI CHANGE KAR SAKTA HAI!
    int currentGear;
```

```
void StartEngine() {
        isEngineOn = true;
        cout << "Engine started!" << endl;
    }

    // ... other methods
};

int main() {
    SportsCar myCar;
    myCar.brand = "Ford";
    myCar.model = "Mustang";

    myCar.StartEngine();
    myCar.currentSpeed = 500;  // ❌ DANGER! Impossible speed set kar diya
    cout << "Current speed: " << myCar.currentSpeed; // 500?? Seriously??

    return 0;

}
```

✅ Code Example: With Encapsulation (Solution)

cpp

```
class SportsCar {

private:    // ✅ DATA SECURITY - Ab directly access nahi ho sakta

    string brand;

    string model;

    bool isEngineOn;

    int currentSpeed;    // ✅ PRIVATE - secure!

    int currentGear;

    string tireCompany;



public:

    // Constructor
```

```
    SportsCar(string b, string m) : brand(b), model(m),   isEngineOn(false ,
currentSpeed(0), currentGear(0),tireCompany("MRF") {}

    // Public methods - interface

    void StartEngine() {

        isEngineOn = true;

        cout << brand << " " << model << " engine started!" << endl;

    }

      void ShiftGear(int gear) {

        if(!isEngineOn) {

            cout << "Engine is off! Cannot shift gear." << endl;

            return;

        }

        currentGear = gear;

        cout << "Shifted to gear " << gear << endl;

    }

      void Accelerate() {

        if(!isEngineOn) {

            cout << "Start engine first!" << endl;

            return;

        }

        currentSpeed += 20;

        cout << "Accelerating to " << currentSpeed << " km/hr" << endl;

    }

     void ApplyBrake() {

        currentSpeed -= 20;
```

```
    if(currentSpeed < 0) currentSpeed = 0;
    cout << "Braking... Current speed: " << currentSpeed << " km/hr" << endl;
}

void StopEngine() {
    isEngineOn = false;
    currentSpeed = 0;
    currentGear = 0;
    cout << "Engine turned off" << endl;
}

// ✅ GETTERS - Sirf value read karne ke liye
int getCurrentSpeed() {
    return currentSpeed;
}

string getTireCompany() {
    return tireCompany;
}

// ✅ SETTERS - Validation ke saath value set karne ke liye
void setTireCompany(string newTire) {
    // Yahan validation kar sakte hain
    if(isValidTire(newTire)) {
```

```
            tireCompany = newTire;
            cout << "Tire company changed to " << newTire << endl;
        } else {
            cout << "Invalid tire company!" << endl;
        }
    }

private:
    bool isValidTire(string tire) {
        // Validation logic here
        return !tire.empty();  // Simple validation for example
    }
};
```

✅ Usage in Main Function with Encapsulation:

cpp

```
int main() {
    SportsCar myCar("Ford", "Mustang");

    myCar.StartEngine();
    myCar.ShiftGear(1);
    myCar.Accelerate();
    // ❌ myCar.currentSpeed = 500;  // AB YE ERROR DEGA! Compile time error
    // ✅ Sirf getter se speed check kar sakte hain
    cout << "Current speed: " << myCar.getCurrentSpeed() << endl;
```

```
    // ✅ Setter se tire change kar sakte hain with validation
    myCar.setTireCompany("Michelin");
    // ❌ Direct access nahi possible
    // myCar.tireCompany = "Random"; // ERROR - private member
     myCar.StopEngine();
     return 0;
}
```

✨ Key Points - Encapsulation:

- ✅ Bundling: Data + Methods ek saath class mein
- ✅ Data Security: Private variables + Public getters/setters
- ✅ Validation: Setters mein validation daal sakte hain
- ✅ Control: Hum control karte hain kaun sa data kahan access ho

💡 Getters & Setters ka Smart Use:
Direct variable public karne se accha getters/setters use karo taki validation aur control ho!

🔷 Abstraction vs Encapsulation - Clear Difference

Aspect	Abstraction 🎭	Encapsulation 📦
Focus	Data Hiding	Data Security
Working	Implementation hide karta hai	Data bundle aur protect karta hai
Real Example	Car drive karna - engine working nahi janna	Car ka odometer - directly change nahi kar sakte

Access Control	N/A	Access modifiers (public, private, protected)
Result	User friendly interface	Secure aur controlled access

🧠 **Analogy:**

- Abstraction: Car drive karna seekhna (internal working nahi janna)
- Encapsulation: Car ke important parts ko lock karna (taki koi chhed-chhad na kar sake)

💡 **Important Note:**

- Abstraction: Agar aap data dekh bhi lo, koi farak nahi padta (engine dekh sakte ho)
- Encapsulation: Agar aap data access kar lo, bahut farak padta hai (odometer change kar doge)

🔶 Access Modifiers - Data Security ke Lie

📌 3 Types of Access Modifiers:

Public 🌐

- Koi bhi access kar sakta hai
- Example: Car ka AC temperature - koi bhi set kar sakta hai

Private 🔒

- Sirf class ke andar access
- Example: Car ki current speed - directly change nahi kar sakte

Protected 🛡️

- Class aur uski child classes access kar sakti hain
- Inheritance mein use hoga (next video)

✅ Code Example Access Modifiers:

cpp

```cpp
class Car {
public:     // ✅ Koi bhi access kar sakta hai
    void StartEngine() { /* anyone can call */ }
```

```
private:    // ✅ Sirf isi class mein access
    int currentSpeed;    // Directly change nahi ho sakti

protected:  // ✅ Is class aur child classes access kar sakti hain
    string engineType;   // Inheritance mein useful hoga

};
```

📚 POWERFUL SUMMARY - Ek Nazar Mein Poora Content

🎯 Programming Evolution Timeline:

`Machine Language (0,1) → Assembly Language (MOV) → Procedural Programming (C) → OOP`

🚀 OOP Ki 3 Big Benefits:

- Real World Modeling 🌍 - Objects aur unke interactions
- Data Security 🔒 - Private data, controlled access
- Scalability & Reusability 📈 - Complex apps easily banaye

🔑 Object Ke 2 Important Parts:

- Characteristics 🏷️ (Brand, Model, Engine Status)
- Behaviors 💪 (Start, Stop, Accelerate, Brake)

🏛️ OOP Ke 4 Pillars:

- Abstraction 💪 - Unnecessary details hide karo
- Encapsulation 📦 - Data + Methods bundle karo + secure karo
- Inheritance 👨‍👩‍👧‍👦 - Coming next (Code reuse)
- Polymorphism 💪 - Coming next (Multiple forms)

💎 Abstraction vs Encapsulation:

Abstraction: Goal → *Simplification*, Focus → *Outer behavior*, Example → *Driving a car*

Encapsulation: Goal → *Data protection*, Focus → *Inner implementation*, Example → *Car engine hidden inside body*

🌟 Final Golden Rules:

- "OOP mein socho: Real world jaisi objects banayo, unko aapas mein interact karayo - coding easy ho jayegi!" ✨
- "Abstraction: Kya chahiye - Encapsulation: Kaise protect karna hai" 🎯