Object Oriented Programming Pillars:

🔷 What is Encapsulation?

Encapsulation is one of the fundamental **OOP (Object-Oriented Programming)** concepts in C++. It is the mechanism of **hiding data** (variables) and **restricting direct access** to them from outside the class. Instead, data can only be accessed or modified using **public methods (getters & setters)**.

🔑 **Key Idea:**

👉 **Data Hiding** + **Controlled Access** = **Encapsulation**

🔷 Why Use Encapsulation?

✅ **Data Security** – Prevents accidental modifications.

✅ **Code Reusability** – Encapsulated code can be reused easily.

✅ **Data Integrity** – Ensures only valid data is assigned.

✅ **Better Maintenance** – Changes in implementation do not affect other parts of the program.

🔷 How to Implement Encapsulation in C++?

Encapsulation is implemented using **classes** with:

1. **Private Data Members** (cannot be accessed directly).
2. **Public Member Functions** (to access & modify private data).

🔷 Example 1: Encapsulation Using Getters & Setters

```
#include <iostream>
using namespace std;
```

```cpp
class Student {
private:
    string name;
    int age;

public:
    // Setter method to set data
    void setData(string n, int a) {
        name = n;
        if (a >= 0) {
            age = a;
        } else {
            cout << "Invalid age!" << endl;
        }
    }

    // Getter method to get name
    string getName() {
        return name;
    }

    // Getter method to get age
    int getAge() {
        return age;
    }
};

int main() {
    Student s1;
    s1.setData("Alice", 20);

    cout << "Name: " << s1.getName() << endl;
    cout << "Age: " << s1.getAge() << endl;

    return 0;
}
```

📝 **Output**

```
Name: Alice
Age: 20
```

🔍 **Explanation:**

✔️ `name` and `age` are **private**, so they cannot be accessed directly.

✔️ Public methods `setData()`, `getName()`, and `getAge()` allow controlled access.

🔷 Example 2: Encapsulation in Real-Life Scenario

🚗 Car Speed Control System

```
#include <iostream>
using namespace std;

class Car {
private:
    int speed;

public:
    // Constructor
    Car() { speed = 0; }

    // Setter to set speed with validation
    void setSpeed(int s) {
        if (s >= 0 && s <= 200) {
            speed = s;
        } else {
            cout << "Invalid speed!" << endl;
        }
    }

    // Getter to get speed
    int getSpeed() {
        return speed;
    }
```

```
};

int main() {
    Car myCar;
    myCar.setSpeed(150);

    cout << "Car Speed: " << myCar.getSpeed() << " km/h" << endl;

    myCar.setSpeed(250);  // Invalid speed, will not update

    return 0;
}
```

📝 **Output**

```
Car Speed: 150 km/h
Invalid speed!
```

🔷 Advantages of Encapsulation

🔷 **Protects Data** – Prevents direct modification of private members.

🔷 **Increases Flexibility** – Data can be modified with conditions.

🔷 **Enhances Code Readability** – Clear separation of data and functions.

🔷 **Improves Maintainability** – Changes in implementation do not affect other parts.

🔷 Summary

Feature	Description
Encapsulation	Wrapping data & methods in a single unit (class).
Access Modifiers	`private`, `protected`, `public` to control access.
Data Hiding	Prevents direct access to sensitive data.
Getters & Setters	Provide controlled access to private data.

🎯 Student Task: Banking System

💡 **Task Description:**

Create a **Bank Account** system using **C++ encapsulation** where:

1. The **account balance** is private and cannot be accessed directly.
2. Users can **deposit** money, but **only if the amount is positive**.
3. Users can **withdraw** money, but **only if they have sufficient balance**.
4. The system should display the **account holder's name and balance**.

📝 Task Requirements

- Use a **class** named `BankAccount` with private variables:
  - `accountHolder` (string)
  - `balance` (double)
- Implement **getter and setter** functions:
  - `deposit(double amount)` → **Adds money if amount > 0**
  - `withdraw(double amount)` → **Deducts money if balance is sufficient**
  - `getBalance()` → **Returns the account balance**
- Create a **menu-driven program** to interact with the user.

✅ Expected Output

```
Welcome to the Bank System!

Enter Account Holder Name: Alice

Choose an option:
1. Deposit Money
2. Withdraw Money
3. Check Balance
4. Exit

Enter choice: 1
Enter deposit amount: 500
Deposit Successful!
```

```
Enter choice: 2
Enter withdrawal amount: 200
Withdrawal Successful!

Enter choice: 3
Current Balance: 300

Enter choice: 4
Thank you for using our Bank System!
```

🧠 Bonus Challenge

1. Implement **multiple accounts**.
2. Add a **PIN system** for security.
3. Display **transaction history**.

▼ **Solution**

```
#include <iostream>
#include <vector>
using namespace std;

class BankAccount {
private:
    string accountHolder;
    double balance;
    int pin;
    vector<string> transactionHistory;

public:
    // Constructor
    BankAccount(string name, double initialBalance, int pinCode) {
        accountHolder = name;
        pin = pinCode;
        if (initialBalance >= 0)
            balance = initialBalance;
        else {
```

```
        balance = 0;
        cout << "Invalid initial balance. Setting balance to 0." << endl;
    }
}

// PIN Verification
bool verifyPin(int enteredPin) {
    return pin == enteredPin;
}

// Deposit money
void deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        transactionHistory.push_back("Deposited: $" + to_string(amount));
        cout << "Deposit Successful! New Balance: $" << balance << endl
    } else {
        cout << "Deposit amount must be positive!" << endl;
    }
}

// Withdraw money
void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        transactionHistory.push_back("Withdrew: $" + to_string(amount));
        cout << "Withdrawal Successful! New Balance: $" << balance << e
    } else {
        cout << "Insufficient balance or invalid amount!" << endl;
    }
}

// Get account balance
double getBalance() {
    return balance;
}

// Display account details
```

```
    void display() {
        cout << "\nAccount Holder: " << accountHolder << endl;
        cout << "Current Balance: $" << balance << endl;
    }

    // Show transaction history
    void showTransactionHistory() {
        cout << "\nTransaction History for " << accountHolder << ":" << endl
        for (string transaction : transactionHistory) {
            cout << transaction << endl;
        }
    }
};

// Main function with multiple accounts
int main() {
    int numAccounts;
    cout << "Enter the number of bank accounts to create: ";
    cin >> numAccounts;

    vector<BankAccount> accounts;  // Vector to store multiple accounts

    // Creating multiple accounts
    for (int i = 0; i < numAccounts; i++) {
        string name;
        double initialBalance;
        int pin;

        cout << "\nEnter details for Account " << i + 1 << endl;
        cout << "Account Holder Name: ";
        cin.ignore();
        getline(cin, name);
        cout << "Enter Initial Balance: ";
        cin >> initialBalance;
        cout << "Set a 4-digit PIN: ";
        cin >> pin;

        accounts.push_back(BankAccount(name, initialBalance, pin));
```

```
}

int choice, accountIndex, enteredPin;
double amount;

while (true) {
    cout << "\nSelect an account (1-" << numAccounts << "): ";
    cin >> accountIndex;
    if (accountIndex < 1 || accountIndex > numAccounts) {
        cout << "Invalid account selection!" << endl;
        continue;
    }

    accountIndex--; // Adjust for zero-based index
    cout << "Enter PIN: ";
    cin >> enteredPin;

    if (!accounts[accountIndex].verifyPin(enteredPin)) {
        cout << "Incorrect PIN! Try again." << endl;
        continue;
    }

    do {
        // Menu options
        cout << "\nChoose an option:" << endl;
        cout << "1. Deposit Money" << endl;
        cout << "2. Withdraw Money" << endl;
        cout << "3. Check Balance" << endl;
        cout << "4. Show Transaction History" << endl;
        cout << "5. Switch Account" << endl;
        cout << "6. Exit" << endl;
        cout << "Enter choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter deposit amount: ";
                cin >> amount;
```

```
                accounts[accountIndex].deposit(amount);
                break;
            case 2:
                cout << "Enter withdrawal amount: ";
                cin >> amount;
                accounts[accountIndex].withdraw(amount);
                break;
            case 3:
                cout << "Current Balance: $" << accounts[accountIndex].getE
                break;
            case 4:
                accounts[accountIndex].showTransactionHistory();
                break;
            case 5:
                cout << "Switching account..." << endl;
                break;
            case 6:
                cout << "Thank you for using our Bank System!" << endl;
                return 0;
            default:
                cout << "Invalid choice! Please try again." << endl;
            }
        } while (choice != 5);
    }

    return 0;
}
```

C++ Inheritance

1️⃣ What is Inheritance?

- **Inheritance** is a fundamental feature of Object-Oriented Programming (OOP) in C++.
- It allows a class (child/derived class) to **inherit** properties and behaviors (variables & methods) from another class (parent/base class).

- This promotes **code reusability** and **hierarchical relationships**.

2️⃣ Why Use Inheritance?

✅ **Code Reusability** – Avoid rewriting common code in multiple classes.

✅ **Hierarchy Representation** – Helps in structuring code using parent-child relationships.

✅ **Extensibility** – Allows easy modifications and enhancements.

✅ **Polymorphism Support** – Enables method overriding and dynamic method binding.

3️⃣ Types of Inheritance

C++ supports five types of inheritance:

Type	Description
Single Inheritance	A single derived class inherits from a single base class.
Multiple Inheritance	A derived class inherits from more than one base class.
Multilevel Inheritance	A derived class acts as a base class for another derived class.
Hierarchical Inheritance	Multiple derived classes inherit from a single base class.
Hybrid (Virtual) Inheritance	Combination of multiple and hierarchical inheritance to prevent ambiguity using virtual base class.

4️⃣ Syntax of Inheritance

```
class Parent {
    // Base class members
};

class Child : access_specifier Parent {
    // Derived class members
};
```

- **Access Specifier**: `public`, `private`, or `protected`.

5️⃣ Access Specifiers in Inheritance

🔷 How Access Specifiers Affect Inherited Members:

Base Class Member	Public Inheritance	Protected Inheritance	Private Inheritance
public members	remain `public` in derived class	become `protected`	become `private`
protected members	remain `protected`	remain `protected`	become `private`
private members	**NOT inherited**	**NOT inherited**	**NOT inherited**

Example:

```
class Parent {
public:
    int a;
protected:
    int b;
private:
    int c;  // Not inherited
};

class Child : public Parent {
    // a remains public
    // b remains protected
    // c is not accessible
};
```

6️⃣ Single Inheritance

- **One base class → One derived class.**

```
#include <iostream>
using namespace std;

class Animal {
public:
```

```
    void eat() { cout << "This animal eats food." << endl; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Dog barks." << endl; }
};

int main() {
    Dog d;
    d.eat();  // Inherited from Animal
    d.bark();
    return 0;
}
```

7️⃣ Multiple Inheritance

- **One child class inherits from multiple base classes.**

```
#include <iostream>
using namespace std;

class Parent1 {
public:
    void show1() { cout << "Base Class 1" << endl; }
};

class Parent2 {
public:
    void show2() { cout << "Base Class 2" << endl; }
};

class Child : public Parent1, public Parent2 {
public:
    void show3() { cout << "Derived Class" << endl; }
};

int main() {
```

```
    Child obj;
    obj.show1();
    obj.show2();
    obj.show3();
    return 0;
}
```

8️⃣ Multilevel Inheritance

- **A class inherits from a derived class** (i.e., Grandparent → Parent → Child).

```
#include <iostream>
using namespace std;

class Grandparent {
public:
    void grandparentFunction() { cout << "This is the grandparent class." <<
endl; }
};

class Parent : public Grandparent {
public:
    void parentFunction() { cout << "This is the parent class." << endl; }
};

class Child : public Parent {
public:
    void childFunction() { cout << "This is the child class." << endl; }
};

int main() {
    Child c;
    c.grandparentFunction();
    c.parentFunction();
    c.childFunction();
    return 0;
}
```

9️⃣ Hierarchical Inheritance

- **One base class → Multiple derived classes.**

```
#include <iostream>
using namespace std;

class Parent {
public:
    void display() { cout << "This is the parent class." << endl; }
};

class Child1 : public Parent {
public:
    void show1() { cout << "Child1 class function." << endl; }
};

class Child2 : public Parent {
public:
    void show2() { cout << "Child2 class function." << endl; }
};

int main() {
    Child1 obj1;
    obj1.display();
    obj1.show1();

    Child2 obj2;
    obj2.display();
    obj2.show2();

    return
}
```

📖 Polymorphism

✅ Definition:

The word **Polymorphism** is derived from Greek — **"Poly"** means *many* and **"Morph"** means *forms*.

Polymorphism in C++ means **one function or operator behaves differently in different situations**.

✅ Types of Polymorphism:

Type	Compile-Time (Static)	Run-Time (Dynamic)
Definition	Decision made at compile-time	Decision made at run-time
Examples	Function Overloading, Operator Overloading	Function Overriding, Virtual Functions
Speed	Faster	Slower (due to virtual table)
Usage	More general	For dynamic behavior

✅ 1. Compile-Time Polymorphism (Static Binding / Early Binding):

👉 Resolved during the **compilation phase**.

🔶 A. Function Overloading

- Same function name, different parameter list.
- Decided based on arguments passed.

Example:

```
#include<iostream>
using namespace std;

class Calculator {
public:
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
```

```
    }
};

int main() {
    Calculator c;
    cout << c.add(5, 3) << endl;       // 8
    cout << c.add(5.5, 3.3) << endl;   // 8.8
}
```

🔶 B. Operator Overloading

- Redefining operators to work with user-defined objects.

Example:

```
#include<iostream>
using namespace std;

class Complex {
public:
    int real, imag;
    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator + (Complex const &obj) {
        return Complex(real + obj.real, imag + obj.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;
    c3.display();  // 4 + 6i
}
```

✅ 2. Run-Time Polymorphism (Dynamic Binding / Late Binding):

👉 Resolved **during run-time** using **virtual functions**.

🔶 Function Overriding

- Same function name and parameters in both base and derived class.
- Achieved using **virtual functions**.

💻 What is a Virtual Function?

✅ Definition (In simple words):

A **virtual function** is a function that is **declared in the base class** but **re-defined (overridden)** by the **derived class**.

It allows us to **call the correct function based on the object type** — even if we are using a pointer or reference of the base class.

🎯 Why Do We Need Virtual Functions?

Imagine you are playing a game with different animals, and you press a button to hear their sound.

- 🐶 **Dog barks** → Woof Woof
- 🐱 **Cat meows** → Meow Meow
- 🐮 **Cow moos** → Moo Moo

Now, if you have a pointer of type **Animal** and point it to **Dog**, you should hear **"Woof Woof"** not some default animal sound, right?

This is what a **virtual function** helps you achieve — calling the correct function based on the **real object**.

📝 Basic Example Without Virtual Function:

```
#include<iostream>
using namespace std;
```

```
class Animal {
public:
    void sound() {
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* a;
    Dog d;
    a = &d;
    a->sound();  // Output: Animal makes a sound
}
```

👉 **Why?** Because `a` is a pointer of type **Animal**, so it calls the **Animal's sound()** function, NOT Dog's.

✅ Now, Let's Add Virtual Keyword:

```
#include<iostream>
using namespace std;

class Animal {
public:
    virtual void sound() {    // Made it virtual
        cout << "Animal makes a sound" << endl;
    }
};

class Dog : public Animal {
public:
```

```
    void sound() override {
        cout << "Dog barks" << endl;
    }
};

int main() {
    Animal* a;
    Dog d;
    a = &d;
    a→sound();  // Output: Dog barks
}
```

🎉 Now It Works Correctly!

Because of the **virtual keyword**, the program knows it should check the **real object** (Dog) and call **Dog’s sound()**.

✅ What Actually Happens?

With `virtual`	Without `virtual`
Looks at the real object	Looks at pointer type
Calls `Dog::sound()`	Calls `Animal::sound()`
Supports Polymorphism	Does not support Polymorphism

✅ Where Do We Use Virtual Functions in Real Life?

1. **Games:** Different characters have different actions.
2. **Banking App:** Different types of accounts (Saving, Current) have different interest calculations.
3. **E-commerce:** Different product types may have different shipping costs.

✅ Rule to Remember:

- Virtual functions are used when you **override a function in child classes** and want to call the correct version based on the object, not the pointer type.

- Helps us achieve **Runtime Polymorphism** (decisions made during program running).

✅ Syntax:

```
class Base {
public:
    virtual void display() {
        cout << "Base display" << endl;
    }
};
```

✅ Final Example:

```
class Animal {
public:
    virtual void sound() { cout << "Animal sound" << endl; }
};

class Cat : public Animal {
public:
    void sound() override { cout << "Cat meows" << endl; }
};

int main() {
    Animal* a;
    Cat c;
    a = &c;
    a->sound();  // Output: Cat meows
}
```

✅ Real-Life Example of Polymorphism:

🔷 Shape (Base Class):

`area()` function behaves differently based on the shape.

Derived Class	Behavior of `area()`
Circle	Calculates area of circle
Rectangle	Calculates area of rectangle
Triangle	Calculates area of triangle

```
class Shape {
public:
    virtual void area() {
        cout << "Calculating area of shape" << endl;
    }
};

class Circle : public Shape {
public:
    void area() override {
        cout << "Area of Circle: πr^2" << endl;
    }
};

class Rectangle : public Shape {
public:
    void area() override {
        cout << "Area of Rectangle: l*b" << endl;
    }
};
```

✅ Advantages of Polymorphism:

✔ Code Reusability

✔ Easy Maintenance

✔ Flexibility and Extensibility

✔ Reduces Code Complexity

✔ Supports Interface Design

✅ Difference Between Overloading and Overriding:

Feature	Function Overloading	Function Overriding
Binding	Compile-time	Run-time
Inheritance	Not required	Required
Parameters	Must differ	Must be same
Keyword	No special keyword	Needs `virtual` in base class
Purpose	Increase readability	Redefine behavior

✅ Quick Revision Diagram:

```
Polymorphism
│
├── Compile-Time (Static)
│     ├── Function Overloading
│     └── Operator Overloading
│
└── Run-Time (Dynamic)
      ├── Function Overriding
      └── Virtual / Pure Virtual Functions
```

🚀 Student Task 1:

Problem Statement: Online Payment Gateway System

You are tasked with developing a basic **Online Payment Gateway** that allows users to pay through different payment methods like **Credit Card**, **Debit Card**, **UPI**, and **Net Banking**.

Your system should use **polymorphism** to handle the payment, where the base class has a function to process payment, and each payment method class overrides it to show how the payment is processed differently.

👉 **Key Requirements:**

- Create a base class `PaymentMethod` with a virtual function `processPayment()`.
- Create child classes: `CreditCard`, `DebitCard`, `UPI`, `NetBanking`.
- Override `processPayment()` in each child class to display how the payment is processed.

- The user can choose the payment method at runtime (Runtime Polymorphism).
- Use pointers or references to demonstrate **polymorphism**.

🎯 Sample Input (via function calls, simulate runtime behavior):

```
PaymentMethod *payment;

payment = new CreditCard();
payment→processPayment();

payment = new UPI();
payment→processPayment();

payment = new NetBanking();
payment→processPayment();
```

✅ Expected Output:

```
Processing payment using Credit Card...
Credit Card Number verified. Payment Successful!

Processing payment using UPI...
UPI ID verified. Payment Successful!

Processing payment using Net Banking...
Net Banking credentials verified. Payment Successful!
```

💡 Hints for Students:

- Use **virtual functions** for runtime polymorphism.
- Think about real-world payment systems and how each method needs different verifications.
- Add your own messages or functionalities like transaction ID generation, verification steps, etc., to make it realistic.

📌 Learning Outcome:

- Understand the power of **polymorphism** in handling different objects through a common interface.
- Learn how real-world systems like **Paytm, Google Pay, Razorpay** handle multiple payment modes dynamically.

🚀 Student Task 2:

Problem Statement: Vehicle Rental System

You are hired to build a **Vehicle Rental System** where customers can rent different types of vehicles like **Car**, **Bike**, or **Truck**.

Each vehicle has its own way of calculating the rental charges based on the number of days rented.

Your system should demonstrate **runtime polymorphism** where the base class `Vehicle` has a virtual function `calculateRental(int days)`, and each derived class (Car, Bike, Truck) overrides it to calculate charges differently.

✅ Requirements:

- Create a base class `Vehicle` with a pure virtual function `calculateRental(int days)`.
- Create derived classes `Car`, `Bike`, and `Truck`.
- Each class has a different daily rental rate:
    - Car: ₹500 per day
    - Bike: ₹200 per day
    - Truck: ₹1000 per day
- Calculate total rental charges based on the number of days.
- Use **polymorphism** to calculate and display the charges.

🎯 Sample Input (via function calls):

```
Vehicle* vehicle;

vehicle = new Car();
vehicle→calculateRental(4);  // Renting Car for 4 days
```

```
vehicle = new Bike();
vehicle->calculateRental(5);  // Renting Bike for 5 days

vehicle = new Truck();
vehicle->calculateRental(3);  // Renting Truck for 3 days
```

✅ Expected Output:

```
Car rented for 4 days. Total Charges: ₹2000
Bike rented for 5 days. Total Charges: ₹1000
Truck rented for 3 days. Total Charges: ₹3000
```

💡 Hints for Students:

- Use **virtual functions** to implement runtime polymorphism.
- Think of how different vehicles have different rates in real life.
- You can extend the system by adding features like driver charges, fuel charges, etc.

📌 Learning Outcome:

- Strong hands-on understanding of **polymorphism and virtual functions** in C++.
- Real-world simulation of a system like **Zoomcar, Ola Rentals, or Bike/Bike Rentals**.

📖Abstraction

✅ Definition:

Abstraction is a process of **hiding internal implementation details** and **showing only essential features** to the user.

> 🎯 "Show what is necessary, hide the complexity!"

📌 Example in Real Life:

- **Car**: You know how to drive a car using the steering wheel, accelerator, and brakes.
- You don't need to know how the engine or fuel system works internally — that's **abstraction**.

✅ Key Points:

✔ Focuses on **what an object does**, not **how it does it**.

✔ Achieved using **abstract classes** and **interfaces** (pure virtual functions).

✔ Helps in **security** by hiding implementation details.

✅ How is Abstraction Achieved in C++?

Method	Description
Abstract Class	A class containing at least one **pure virtual function**.
Pure Virtual Function	`virtual void show() = 0;` Forces the derived class to override.
Interface-like behavior	Created using abstract class with only pure virtual functions.

✅ Pure Virtual Function:

- Syntax: `virtual returnType functionName() = 0;`
- Has **no definition** in the base class.
- Makes the class **abstract**.

```
class Shape {
public:
    virtual void area() = 0;  // Pure virtual function
};
```

✅ Abstract Class:

- A class that **cannot be instantiated** directly.
- Contains **at least one pure virtual function**.

- Acts as a **blueprint** for derived classes.

```
class Shape {
public:
    virtual void area() = 0;  // Pure virtual function
};
```

✅ Example of Abstraction in C++:

```
#include<iostream>
using namespace std;

class Shape {
public:
    virtual void area() = 0;  // Pure virtual function
};

class Circle : public Shape {
    int radius;
public:
    Circle(int r) : radius(r) {}
    void area() override {
        cout << "Area of Circle: " << 3.14 * radius * radius << endl;
    }
};

class Rectangle : public Shape {
    int length, breadth;
public:
    Rectangle(int l, int b) : length(l), breadth(b) {}
    void area() override {
        cout << "Area of Rectangle: " << length * breadth << endl;
    }
};

int main() {
    Shape* s;
```

```
    Circle c(5);
    Rectangle r(4, 6);

    s = &c;
    s->area();

    s = &r;
    s->area();

    return 0;
}
```

✅ Output:

```
Area of Circle: 78.5
Area of Rectangle: 24
```

✅ Benefits of Abstraction:

Benefit	Description
✅ **Security**	Hides implementation details
✅ **Code Reusability**	Can reuse abstract class for different objects
✅ **Easy Maintenance**	Changes in implementation don't affect the interface
✅ **Improved Flexibility**	Different classes can implement the same abstract class
✅ **Focus on essential**	Keeps the program simple and easy to understand

---

✅ Real-Life Examples of Abstraction:

Example	What you see (Abstracted)	What is hidden (Implementation)
ATM Machine	Deposit, Withdraw, Balance check	Internal banking process
Car	Steering, Brake, Accelerator	Engine, Fuel system
Mobile Phone	Touchscreen UI	Circuitry, Chips

✅ Diagram Representation:

```
Abstract Class: Shape
 |
 ├──> Circle : area() defined
 |
 └──> Rectangle : area() defined
```

✅ Important Notes for Interviews & Exams:

Term	Description
Abstraction	Hides complexity, shows only essential details
Abstract Class	Has at least one pure virtual function
Pure Virtual Function	`= 0` forces derived class to override
Interface (C++ style)	Abstract class with only pure virtual functions
Usage	Security, code maintenance, extensibility

✅ Difference Between Abstraction and Encapsulation:

Feature	**Abstraction**	**Encapsulation**
Definition	Hiding implementation details	Hiding data using access specifiers
Focus	What an object does	How data is protected
Achieved by	Abstract class, Pure virtual functions	Access specifiers (private, protected, public)
Example	ATM UI	Class with private variables

Student Task 1:

📝 Problem Statement:

Design an **Online Shopping Delivery System** where there are multiple delivery partners like **BlueDart**, **Delhivery**, and **IndiaPost**.

Each delivery partner follows its unique process for handling deliveries. However, the user should just call `startDelivery()` without knowing the internal steps of each courier company.

👉 **Hint:** Use **Abstraction** — Create an abstract base class `CourierService` and derive specific classes like `BlueDart`, `Delhivery`, and `IndiaPost`.

✅ Expected Input/Output Example:

Input:

Create objects of BlueDart, Delhivery, and IndiaPost and call `startDelivery()`.

Output:

```
BlueDart: Packing the parcel, scanning, dispatching via air cargo.
Delhivery: Packing the parcel, assigning to nearest hub, dispatching.
IndiaPost: Packing the parcel, sending to the postal sorting center, dispatch
ing.
```

Happy Coding!