

Lecture 2 : OOPS (Abstraction & Encapsulation)

1. Why Did We Move Beyond Procedural Programming?

1.1 Early Languages

1. Machine Language (Binary)

- Direct CPU instructions in 0s & 1s.
- **Drawbacks:**
 - Extremely error-prone: one bit flip breaks the program.
 - Tedious to write and maintain.
 - No abstraction—every detail is manual.

2. Assembly Language

- Introduced mnemonics (e.g. `MOV A, 61h`) instead of raw bits.
- **Still hardware-tied:** code changes with CPU architecture.
- **Scalability:** remains very limited for large systems.

1.2 Procedural (Structured) Programming

- **Features Introduced:**
 - **Functions** for code reuse
 - **Control structures:** `if-else`, `switch`, `for/while` loops
 - **Blocks** for grouping statements
- **Advantages:**
 - Improved readability over assembly.
 - Modularized small to mid-size programs.
- **Limitations:**
 - **Poor real-world mapping:** Difficult to model complex entities (e.g. a ride-booking system's users, drivers, payments).
 - **Data security gaps:** No built-in access control—everything is globally visible.
 - **Reusability & scalability:** Functions alone can't enforce consistent interfaces or safe extension.

2. Entering Object-Oriented Programming

- **Core Idea:** Model your application as **interacting objects** mirroring real-world entities.
- **Benefits:**
 - **Natural mapping** of domain concepts (User, Car, Ride).
 - **Secure data encapsulation**—control who can read or modify state.
 - **Code reuse** via inheritance and interfaces.
 - **Scalability** through loosely coupled modules.

3. Modeling Real-World Entities in Code

3.1 Objects, Classes, & Instances

- **Object:** A real-world “thing” with attributes and behaviors.
- **Class:** Blueprint defining those attributes (fields) and behaviors (methods).
- **Instance:** Concrete object in memory, created via the class.

4. Deep Dive: Pillar 1 – Abstraction

Definition:

Abstraction hides unnecessary implementation details from the **client** and **exposes only what is essential to use an object's functionality**.

4.1. Real-World Analogies

- **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.
- **Using a TV or Laptop**
 - **What you do:** Press buttons on a remote or click icons.
 - **What you don't need to know:** How the display panel refreshes, how the CPU executes machine code, how the OS schedules tasks.
 - **Abstraction in action:** A graphical interface abstracts away thousands of low-level operations.

4.2. Language-Level Abstraction

- **Control Structures as Abstraction**
 - Keywords like **if**, **for**, **while** let you express complex branching and loops without writing jump addresses or machine instructions.
 - The compiler translates these high-level constructs into assembly or machine code behind the scenes.

5. Code-Based Abstraction: Abstract Classes & Interfaces

5.1 Abstract Class Example (C++)

```
// Abstract interface for any Car type
class Car {
public:
    // Pure virtual methods - no implementation here
    virtual void startEngine() = 0;
    virtual void shiftGear(int newGear) = 0;
    virtual void accelerate() = 0;
    virtual void brake() = 0;
    virtual ~Car() {}
};
```

- **Key Points**

- The `Car` class declares *what* operations must exist but hides *how* they work.
- No code for `startEngine()`, etc., lives here—only signatures.
- Clients use `Car*` pointers without needing concrete details.

5.2 Concrete Subclass Example

// See Code section for full Code example

6. Benefits of Abstraction

1. **Simplified Interfaces:** Clients focus on *what* an object does, not *how* it does it.
2. **Ease of Maintenance:** Internal changes (e.g., switching from a V6 to an electric motor) don't affect client code.
3. **Code Reuse:** Multiple concrete classes can implement the same abstract interface (e.g., `SportsCar`, `SUV`, `ElectricCar`).
4. **Reduced Complexity:** Large systems are easier to reason about when broken into abstract modules.

7. Deep Dive: Pillar 2 – Encapsulation

Definition:

Encapsulation bundles an object's data (its state) and the methods that operate on that data into a single unit, and controls access to its inner workings.

7.1. Two Facets of Encapsulation

1. **Logical Grouping**

- Data (fields) and behaviors (methods) that belong together live in the same “capsule” (class).

- Example: A `Car` class encapsulates `engineOn`, `currentSpeed`, `shiftGear()`, `accelerate()`, etc., in one place.

2. Data Security

- Restrict direct external access to sensitive fields to prevent invalid or unsafe operations.
- Example: You can *read* the car's odometer but cannot directly set it back to zero.

7.2. Real-World Analogies

- **Medicine Capsule**
 - The capsule holds both the medicine (data) and its protective shell (access control).
 - You swallow the capsule without exposing its contents directly.
- **Car Odometer**
 - You can view the mileage but *cannot* tamper with it via the dashboard interface.

// See Code section for full Code example

7.3 Access Modifiers in C++

- **public**: Members are accessible everywhere.
- **private**: Members accessible only within the class itself.
- **protected**: Accessible in the class and its subclasses (for inheritance scenarios).

7.4. Getters & Setters with Validation

- **Purpose**: Allow controlled mutation with checks, rather than exposing fields blindly.

7.5. Encapsulation Benefits

1. **Robustness**: Prevents accidental or malicious misuse of internal state.
2. **Maintainability**: Internal changes (e.g., adding new constraints) do not ripple into client code.
3. **Clear Contracts**: Clients interact only via well-defined methods (the public API).
4. **Modularity**: Code is organized into self-contained units, easing testing and reuse.