Lecture 2: OOPS in Java (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.
    - Tedious to write and maintain.
    - No abstraction.

2. Assembly Language

- Introduced mnemonics (e.g., MOV A, 61h).
- Still hardware-tied.
- Limited scalability for large systems.

1.2 Procedural (Structured) Programming

- **Features:**
    - Functions for code reuse
    - Control structures: if-else, switch, loops
    - Grouping statements into blocks
- **Advantages:**
    - Improved readability
    - Modular design
- **Limitations:**
    - Poor real-world modeling
    - No data security
    - Lacks scalability & extensibility

---

2. Entering Object-Oriented Programming

- **Core Idea:** Model apps as interacting objects mirroring real-world entities.
- **Benefits:**
    - Natural domain mapping (User, Car, Ride)
    - Secure data encapsulation
    - Code reuse via inheritance/interfaces
    - Scalability via modular design

---

3. Modeling Real-World Entities in Java

3.1 Objects, Classes, & Instances

- **Object:** Real-world entity with state & behavior
- **Class:** Blueprint defining state (fields) & behavior (methods)
- **Instance:** Object created from a class

```java
class Car {
    String color;
    int speed;

    void accelerate() {
        speed += 10;
    }

    void brake() {
        speed -= 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.color = "Red";
        myCar.accelerate();
        System.out.println("Speed: " + myCar.speed);
    }
}
```

---

4. Deep Dive: Abstraction

Definition:

Hides internal implementation and exposes only necessary interfaces.

4.1 Real-World Analogies

- **Car:** Start, brake, accelerate without knowing internal mechanism.
- **Laptop:** Click icons without knowing how OS or hardware works.

4.2 Language-Level Abstraction

- **Control Structures:** if, for, while abstract complex jump logic.

5. Code-Based Abstraction: Abstract Classes & Interfaces

5.1 Abstract Class Example in Java

```
abstract class Car {
    abstract void startEngine();
    abstract void shiftGear(int gear);
    abstract void accelerate();
    abstract void brake();
}
```

5.2 Concrete Subclass Example

```
class ElectricCar extends Car {
    void startEngine() {
        System.out.println("Starting electric motor...");
    }

    void shiftGear(int gear) {
        System.out.println("Shifting to gear: " + gear);
    }

    void accelerate() {
        System.out.println("Accelerating...");
    }

    void brake() {
        System.out.println("Braking...");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new ElectricCar();
        myCar.startEngine();
        myCar.accelerate();
    }
}
```

6. Benefits of Abstraction

- Simplified interfaces
- Easier maintenance
- Code reuse across different implementations
- Reduced complexity in large systems

---

7. Deep Dive: Encapsulation

Definition:

Encapsulation bundles state and behavior, controlling access to internal data.

7.1 Two Facets of Encapsulation

1. **Logical Grouping:** Class holds related fields and methods.

```
class Car {
    private boolean engineOn;
    private int speed;

    void startEngine() {
        engineOn = true;
    }

    void accelerate() {
        if (engineOn) speed += 10;
    }
}
```

2. **Data Security:** Restrict access using access modifiers

```
class Car {
    private int odometer = 0;

    public int getOdometer() {
        return odometer;
    }

    public void drive(int distance) {
        if (distance > 0) {
            odometer += distance;
        }
    }
}
```

7.2 Access Modifiers in Java

- **public:** Accessible everywhere
- **private:** Accessible only within the class
- **protected:** Accessible within package & subclasses

7.3 Getters & Setters with Validation

```
class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}
```

7.4 Encapsulation Benefits

- Robustness via controlled access
- Maintainability
- Clear interaction contracts
- Modular, testable code