Encapsulation in Java (OOP's Method)

What is Encapsulation?

Encapsulation is a fundamental concept in Object-Oriented Programming (OOP) that bundles data (variables) and methods (functions) together into a single unit called a class. It hides the internal details of how data is stored and accessed from the outside world, allowing controlled access through public methods.`

Simple Definition: Encapsulation means wrapping your data and the methods that operate on that data inside a class, and hiding the internal implementation from the outside world.

Real-Life Example

Think of a car: -

- **Internal parts** (engine, transmission, fuel system) = Private data
- **Dashboard controls** (accelerator, brake, steering) = Public methods
- You don't need to know how the engine works internally; you just use the controls provided
- The car protects its internal parts from direct access—you can't open the engine while driving.

Why Do We Need Encapsulation? (Benefits and Advantages)

1. Data Protection and Security

Encapsulation protects sensitive data from unauthorized or unintended modifications. Only the class's methods can modify private data.

Example: A bank account balance cannot be directly changed to negative. The method controlling it validates before changing.

2. Data Hiding (Information Hiding)

Users of the class don't need to know how the data is stored or how methods work internally. They only know what methods are available to use.

Example: You don't need to know how `accelerate()` increases speed; you just call it.

3. Improved Maintainability

If the internal implementation changes, external code using the class doesn't break. You can modify the internal logic without affecting users of your class.

Example: If you change how speed is calculated internally, the public method interface remains the same.

4. Better Control Over Data

Using getter and setter methods, you can:

- Apply validation rules
- Ensure data consistency
- Control what data can be read or written.

Example: Prevent gear from being set to invalid values (like 7 or -1).

5. Code Reusability

Encapsulated classes can be reused in different programs and projects without exposing internal logic.

6. Easy Testing

Code designed with encapsulation is easier to unit test because each class has clear responsibilities.

7. Flexibility

You can make variables read-only (only getter) or write-only (only setter) as per requirements.

How Does Encapsulation Work?

Encapsulation works through Access Modifiers in Java. There are four main access levels:

1. Private (`private`)

- Access: Only accessible within the same class
- Usage: Hide internal data and helper methods
- Most restrictive level

`private int speed;` // Cannot be accessed outside the class
`private String model;` // Hidden from external code

2. Public (`public`)

- Access: Accessible from anywhere (any class, any package)
- Usage: Methods that you want other classes to use
- Least restrictive level

```
public void accelerate(int x) {   // Can be called from anywhere
// code
}
```

3. Protected (`protected`)

- Access: Accessible within the same class, subclasses, and same package
- Usage: Methods meant for child classes to inherit
- Intermediate level

4. Default (no modifier)

- Access: Only accessible within the same package
- Usage: When you want package-level access
- Intermediate level

For Encapsulation, we primarily use: `private` for data + `public` for methods

---

Implementation Pattern : Private Fields + Public Methods

The standard encapsulation pattern is:

Car Class
Private (Hidden)
-speed
-gear
PUBLIC (Exposed)
- accelerate()
- applyBrake()
- changeGear()
- getGear()
- getSpeed()

```
---
## How to Access Private Data: Getter and Setter Methods
Since private fields cannot be accessed directly, we use **getter** and **setter**
methods.
### Getter Methods
- **Purpose:** Read the private data
- **Access:** Public, returns the private variable
- **Convention:** Starts with `get`, followed by field name
```

```
// Getter for private field 'speed'
public int getSpeed() {
return speed;    // Return the private field value

}
```

```
// Getter for private field 'gear'
public int getGear() {
return gear;     // Return the private field value
}
```

How to use:

```
Car c1 = new Car();
int currentSpeed = c1.getSpeed();    // Access private 'speed' through getter
System.out.println(currentSpeed);    // Output: 0
```

Setter Methods

- Purpose: Modify the private data with validation
- Access: Public, takes a parameter to set the private variable
- Convention: Starts with `set`, followed by field name

```
// Setter for private field 'speed' (with validation)
public void setSpeed(int newSpeed) {
    if (newSpeed >= 0) {
    speed = newSpeed;
    }
}
```

```
// Setter for private field 'gear' (with validation)
public void setGear(int newGear) {
    if (newGear >= 1 && newGear <= 6) {
    gear = newGear;
    }
}
```

How to use:

```
Car c1 = new Car();
c1.setSpeed(100);    // Modify private 'speed' through setter (if valid)
c1.setGear(3);         // Modify private 'gear' through setter (if valid)
```

Real Example: Car Class with Full Encapsulation

```
class Car {
    // STEP 1: Declare private fields (data hiding)
    private int speed;
    private int gear;

    // STEP 2: Constructor (initialize private fields)
    public Car() {
        speed = 0;
        gear = 1;
    }

    // STEP 3: Getter methods (read-only access to private data)
    public int getSpeed() {
        return speed;
    }

    public int getGear() {
        return gear;
    }

    // STEP 4: Setter/Action methods (controlled modification with validation)
    public void accelerate(int x) {
        if (x > 0) {
            speed += x;  // Validation: only positive values
        }
    }

    public void applyBrake(int x) {
        if (x > 0) {
            speed -= x;
            if (speed < 0) {
                speed = 0;  // Validation: speed cannot go below 0
            }
        }
    }

    public void changeGear(int g) {
        if (g >= 1 && g <= 6) {  // Validation: gear must be 1-6
            gear = g;
        }
```

```
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Car c1 = new Car();

        // Access private data through getter (read)
        System.out.println("Initial Speed: " + c1.getSpeed());  // 0
        System.out.println("Initial Gear: " + c1.getGear());    // 1

        // Modify private data through methods (write with validation)
        c1.accelerate(50);  // Speed becomes 50
        c1.changeGear(3);   // Gear becomes 3
        System.out.println("Speed: " + c1.getSpeed());          // 50
        System.out.println("Gear: " + c1.getGear());            // 3

        // Try invalid operations (they are ignored due to validation)
        c1.changeGear(10);  // Invalid gear, ignored
        System.out.println("Gear after invalid change: " + c1.getGear());  // Still 3
    }
}
```

```
**Output:**
 Initial Speed: 0
 Initial Gear: 1
 Speed: 50
 Gear: 3
 Gear after invalid change: 3
```

Comparison: With vs Without Encapsulation

WITHOUT Encapsulation (Bad Practice)

```
class CarBad {
    public int speed;  // Direct access allowed
    public int gear;   // Direct access allowed
}

    // Usage - No validation!
    CarBad c = new CarBad();
```

```
c.speed = -100;   // Invalid! Negative speed set
c.gear = 99;      // Invalid! Gear out of range
```

Problems:

- No validation
- Speed can be negative (illogical)
- Gear can be any number (illogical)
- Data integrity compromised

WITH Encapsulation (Best Practice)

```
class Car {
    private int speed;
    private int gear;

    public int getSpeed() {
        return speed;
    }

    public int getGear() {
        return gear;
    }

    public void setSpeed(int s) {
        if (s >= 0) {          // Validation!
            speed = s;
        }
    }

    public void setGear(int g) {
        if (g >= 1 && g <= 6) {   // Validation!
            gear = g;
        }
    }
}
// Usage - Validation applied!
    Car c = new Car();
    c.setSpeed(-100);     // Rejected, ignored (or throw exception)
    c.setGear(99);        // Rejected, ignored
```

Benefits:

- Validation ensures data integrity
- Invalid operations are prevented
- Speed always remains valid
- Gear always remains in valid range

Key Steps to Implement Encapsulation

1. Declare fields as private

```
private int speed;
private String name;
```

---

2. Provide public getter methods (for reading)

```
public int getSpeed() {
    return speed;
}
```

---

3. Provide public setter / action methods (for writing with validation)

```
public void accelerate(int x) {
    if (x > 0) {
        speed += x;
    }
}
```

---

4. Add validation rules inside methods

```
if (speed < 0) {
    speed = 0;   // Enforce rule
}
```

---

5. Use only public methods to interact with private data

```
Car c1 = new Car();
c1.accelerate(50);    // Use public method

// c1.speed = 100;    //  Never do this! (Compilation error)
```

Summary Table

Aspect	Description	
What	Wrapping data and methods inside a class, hiding details with `private`	
Why	Protect data, ensure validation, improve security and maintainability	
How	Use `private` fields + `public` getter/setter methods	
Access	Read via `getVariable()` method	
Modify	Write via `setVariable()` or action methods with validation	
Benefit	Data integrity, security, control, flexibility, maintainability	
Example	Car class: private speed/gear + public accelerate/brake methods	

Important Takeaways

- Encapsulation = Data + Methods bundled + Hidden implementation
- Private = Hidden (not accessible from outside)
- Public = Visible (accessible from anywhere)
- Always protect your data with `private` and provide controlled access through public methods
- Validation should be inside setter/action methods to ensure data integrity
- Getters = Read-only access, Setters = Write with validation

Encapsulation is the foundation of secure, maintainable, and professional Java code.