



Southeast University

Department of Computer Science and Engineering (CSE)

School of Sciences and Engineering

Semester: (Summer, Year: 2025)

LAB REPORT NO: 04

Course Title: Introduction to Programming Language II (Java) Lab

Course Code: CSE282.2

Batch: 65

**Lab Experiment Name: Object Oriented Programming Concepts
(Encapsulation).**

Student Details

| | Name | ID |
|----|--------------------|---------------|
| 1. | Md. Mahmud Hossain | 2023200000799 |

Submission Date : 10-08-25

Course Teacher's Name : Dr. Mohammed Ashikur Rahman

Lab Report Status

Marks:

Comments:.....

Signature:.....

Date:.....

Lab Task 4: Object Oriented Programming Concepts (Encapsulation).

OBJECTIVES:

- To be familiar with encapsulation
- To be familiar with Get and Set methods

PROBLEM:

1. Create a class called Person with properties such as name, age, gender, address. Include getter and setter methods for each property.
2. Create a class called Employee with properties such as name, id, salary, designation. Include methods to get and set the properties.

Solution:

1.

Problem Analysis:

The task requires designing a class to represent a person with key attributes that are kept private for protection. Encapsulation ensures these attributes are only accessed or changed via specific methods, maintaining data consistency and preventing direct tampering. By adding inheritance, we create a base class for the properties and methods, allowing the Person class to extend it. This approach makes the code more modular and extensible, as the base could potentially be reused for similar entities. User input isn't directly involved here, but the main method demonstrates setting and displaying properties for multiple objects to verify functionality.

Background Theory:

In Java, core OOP principles like access modifiers, encapsulation, getter/setter methods, and inheritance work together to build robust, maintainable code. These concepts control how data is hidden, accessed, and extended across classes.

a. Access Modifiers:

Access modifiers define the scope of visibility for class elements. Java offers four levels:

- Public: Accessible from anywhere.
- Private: Limited to the class itself, ideal for hiding internal data.
- Protected: Available within the class, subclasses, and the same package.
- Default: Accessible only within the same package if no modifier is specified.

b. Encapsulation:

Encapsulation bundles data and operations into a single unit (class) while restricting direct access to internals. Private fields are used, with public methods providing the interface. This safeguards data and allows for validation during modifications.

c. Getter and Setter Methods:

These are public methods for reading (getters) and updating (setters) private fields. Getters return field values, while setters assign new ones, often with checks for validity. Naming follows conventions like `getName()` and `setName(String name)`.

d. Inheritance:

Inheritance allows a subclass to inherit fields and methods from a superclass, promoting reuse and hierarchy. The "extends" keyword is used. Here, it's applied optionally to separate property management into a base class, making the Person class focus on specific behaviors like demonstration in the main method.

These elements combine to create secure, flexible classes that adhere to OOP best practices.

Algorithm Design:

1. Create a base class called "Information" with the following private properties:
 - i. Name
 - ii. Age
 - iii. Gender
 - iv. Address
2. In the base class, implement getter and setter methods for each property:
 - i. For each property, add a getter method to retrieve the current value.
 - ii. For each property, add a setter method that accepts a new value and updates the property.

3. Create a subclass called "Person" that extends "Information".
4. In the Person class's main method, instantiate two Person objects and set their properties using the inherited setter methods:
 - i. Create instances of Person.
 - ii. Use setters to assign values like name, age, gender, and address.
5. Display the details of each object using the inherited getter methods.

Code:

```
class information{
    private String name;
    private int age ;
    private String gender;
    private String address;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getGender() {
        return gender;
    }

    public String getAddress() {
        return address;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

```

    public void setGender(String gender) {
        this.gender = gender;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}

public class Person extends information{

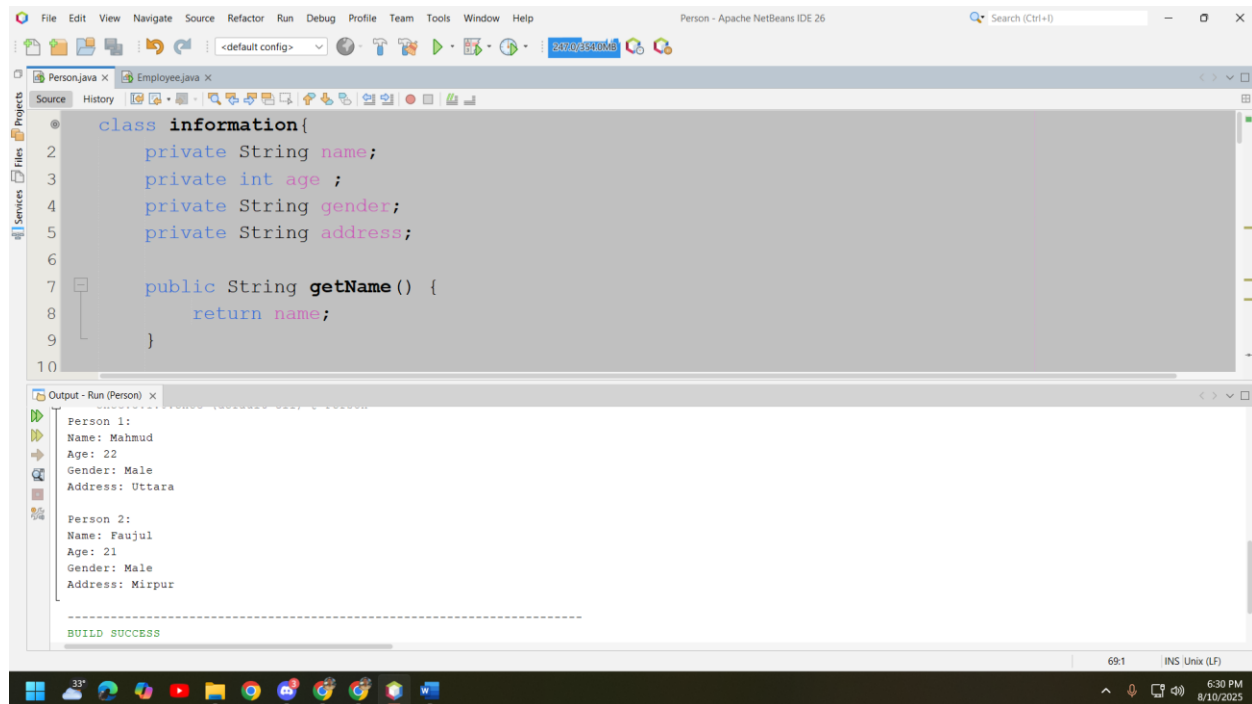
    public static void main(String[] args) {
        Person p1 = new Person();
        p1.setName("Mahmud");
        p1.setAge(22);
        p1.setGender("Male");
        p1.setAddress("Uttara");

        Person p2 = new Person();
        p2.setName("Faujul");
        p2.setAge(21);
        p2.setGender("Male");
        p2.setAddress("Mirpur");

        System.out.println("Person 1:");
        System.out.println("Name: "+p1.getName());
        System.out.println("Age: "+p1.getAge());
        System.out.println("Gender: "+p1.getGender());
        System.out.println("Address: "+p1.getAddress());
        System.out.println("");
        System.out.println("Person 2:");
        System.out.println("Name: "+p2.getName());
        System.out.println("Age: "+p2.getAge());
        System.out.println("Gender: "+p2.getGender());
        System.out.println("Address: "+p2.getAddress());
        System.out.println("");
    }
}

```

Output:



The screenshot displays the Apache NetBeans IDE interface. The main editor window shows a Java class named `information` with the following code:

```
class information{
2   private String name;
3   private int age ;
4   private String gender;
5   private String address;
6
7   public String getName() {
8       return name;
9   }
10 }
```

Below the editor, the 'Output - Run (Person)' window shows the execution results for two persons:

```
Person 1:
Name: Mahmud
Age: 22
Gender: Male
Address: Uttara

Person 2:
Name: Faujul
Age: 21
Gender: Male
Address: Mirpur
```

At the bottom of the output window, it states 'BUILD SUCCESS'. The status bar at the bottom of the IDE indicates '69:1 INS Unix (LF)' and the system clock shows '6:30 PM 8/10/2025'.

Figure 1: Output

2.

Problem Analysis:

The requirement is to model an employee with attributes that must be privately stored to enforce encapsulation. Direct access is avoided, with modifications routed through methods for integrity. Incorporating inheritance, a base class holds the properties and accessors, while the Employee subclass inherits them and uses a main method to create and display objects. This adds a layer of organization, making the code hierarchical and easier to maintain without altering the core encapsulation goal.

Background Theory:

Java's OOP foundation relies on access modifiers, encapsulation, getter/setter methods, and inheritance to manage data securely and efficiently.

a. Access Modifiers:

These control element visibility:

- i. Public: Open to all classes.
- ii. Private: Confined to the declaring class.
- iii. Protected: Accessible in the class, subclasses, and package.
- iv. Default: Package-level access only.

b. Encapsulation:

This principle hides implementation details by privatizing fields and exposing them via methods, ensuring controlled interactions and data protection.

c. Getter and Setter Methods:

Getters fetch private field values, and setters update them, following standard naming (e.g., `getSalary()`, `setSalary(double salary)`). They can include logic for validation.

d. Inheritance:

Through extension, subclasses gain superclass features, reducing duplication. Using "extends," it creates "is-a" relationships. Here, it's employed to abstract property handling into a base, letting Employee inherit and build upon it.

Together, these concepts foster modular, secure code design.

Algorithm Design:

1. Create a base class called "Information" with the following private properties:
 - a. Name
 - b. ID
 - c. Salary
 - d. Designation

2. In the base class, implement getter and setter methods for each property:
 - i. Add getters to return the property values.
 - ii. Add setters to update the properties with provided values.
3. Create a subclass called "Employee" that extends "Information".
4. In the Employee class's main method, instantiate two Employee objects and configure their properties using the inherited setters:
 - i. Create Employee instances.
 - ii. Assign values for name, ID, salary, and designation via setters.
5. Output the object details using the inherited getters.

Code:

```
class information{
    private String name;
    private int id;
    private double salary;
    private String designation;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public double getSalary() {
        return salary;
    }
}
```



```

    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public String getDesignation() {
        return designation;
    }

    public void setDesignation(String designation) {
        this.designation = designation;
    }
}

public class Employee extends information{
    public static void main(String[] args) {
        Employee e1 = new Employee();
        e1.setName("Rakib");
        e1.setId(101);
        e1.setSalary(1000.00);
        e1.setDesignation("Manager");

        Employee e2 = new Employee();
        e2.setName("Faujul");
        e2.setId(201);
        e2.setSalary(2000.00);
        e2.setDesignation("IT specialist");

        System.out.println("Employee 1:");
        System.out.println("Name: "+e1.getName());
        System.out.println("Id: "+e1.getId());
        System.out.println("Salary: "+e1.getSalary());
        System.out.println("Designation: "+e1.getDesignation());
        System.out.println("");
        System.out.println("Employee 2:");
        System.out.println("Name: "+e2.getName());
        System.out.println("Id: "+e2.getId());
        System.out.println("Salary: "+e2.getSalary());
    }
}

```

```

        System.out.println("Designation: "+e2.getDesignation());
        System.out.println("");
    }
}

```

Output:

The screenshot displays the Apache NetBeans IDE interface. The main editor window shows the source code of `Employee.java`. The code defines a class `information` with private attributes `name` (String), `id` (int), `salary` (double), and `designation` (String). It includes a public method `getName()` that returns the `name` attribute. The output window at the bottom shows the execution results of the program. It displays the details for two employees: Employee 1 (Name: Rakib, Id: 101, Salary: 1000.0, Designation: Manager) and Employee 2 (Name: Fauzul, Id: 201, Salary: 2000.0, Designation: IT specialist). The output also indicates a successful build and provides the total execution time and finish date.

```

class information{
    private String name;
    private int id;
    private double salary;
    private String designation;

    public String getName() {
        return name;
    }
}

```

```

--- exec:3.1.0:exec (default-cli) @ Employee ---
Employee 1:
Name: Rakib
Id: 101
Salary: 1000.0
Designation: Manager

Employee 2:
Name: Fauzul
Id: 201
Salary: 2000.0
Designation: IT specialist

-----
BUILD SUCCESS
-----
Total time: 1.006 s
Finished at: 2025-08-10T18:43:08+06:00

```

Figure 2: Output