19CSE204 – OBJECT ORIENTED PARADIGM

LAB PRACTICALS

SUBMITTED BY: **MADHUNISHA M V**

CH.EN.U4CCE22017

SUBMITTED TO: **DR. S. SUTHIR**

ENCAPSULATION

AIM:

To implement and understand the concept of Encapsulation in Java, which is the process of wrapping data (variables) and code (methods) together into a single unit and restricting direct access to some of the object's components using access modifiers and getter/setter methods.

CODE 1:

Untitled | BankApp.java | StudentApp.java

File Edit View

```
class BankAccount {
    private String accountHolder;
    private double balance;

    // Constructor
    BankAccount(String accountHolder, double balance) {
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    // Getter
    public double getBalance() {
        return balance;
    }

    // Setter-like methods for controlled updates
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount + " | New Balance: " + balance);
        } else {
            System.out.println("Invalid deposit amount");
        }
    }

    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount + " | Remaining Balance: " + balance);
        } else {
            System.out.println("Insufficient funds");
        }
    }
}

public class BankApp {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount("Madhunisha", 5000);
        acc.deposit(1000);
        acc.withdraw(2000);
        System.out.println("Final Balance: " + acc.getBalance());
    }
}
```

OUTPUT:

```
C:\Users\kmpvi>cd C:\Users\kmpvi\Documents\JAVA\ENCAPSULATION

C:\Users\kmpvi\Documents\JAVA\ENCAPSULATION>javac BankApp.java

C:\Users\kmpvi\Documents\JAVA\ENCAPSULATION>java BankApp
Deposited: 1000.0 | New Balance: 6000.0
Withdrawn: 2000.0 | Remaining Balance: 4000.0
Final Balance: 4000.0
```

CODE 2:

Untitled | BankApp.java | StudentApp.java

File Edit View

```
class Student {
    private String name;
    private int age;
    private double gpa;

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

    public void setAge(int age) {
        if (age > 0)
            this.age = age;
        else
            System.out.println("Invalid age!");
    }

    public void setGpa(double gpa) {
        if (gpa >= 0 && gpa <= 10)
            this.gpa = gpa;
        else
            System.out.println("Invalid GPA!");
    }

    // Getters
    public String getName() { return name; }
    public int getAge() { return age; }
    public double getGpa() { return gpa; }
}

public class StudentApp {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Manisha");
        s.setAge(21);
        s.setGpa(8.4);
        System.out.println("Student: " + s.getName() + ", Age: " + s.getAge() + ", GPA: " + s.getGpa());
    }
}
```

OUTPUT:

c

```
C:\Users\kmpvi\Documents\JAVA\ENCAPSULATION>javac StudentApp.java

C:\Users\kmpvi\Documents\JAVA\ENCAPSULATION>java StudentApp
Student: Manisha, Age: 21, GPA: 8.4
```

INFERENCE:

The programs illustrate the principle of **Encapsulation**, a key concept of **Object-Oriented Programming (OOP)**.

In the **BankApp** example:

- The class BankAccount has **private data members** (accountHolder and balance) that cannot be accessed directly.
- Controlled access is provided through **public methods** like deposit(), withdraw(), and getBalance().
- This ensures **data protection** by preventing unauthorized or invalid operations (e.g., depositing negative amounts).

In the **StudentApp** example:

- The class Student keeps name, age, and gpa as **private attributes**.
- Access to these fields is managed through **getter and setter methods**, enforcing **validation checks** (like valid age and GPA range).
- This helps maintain **data integrity** and avoids inconsistent states.

Overall, **Encapsulation**:

- Enhances **data security** by hiding internal details.
- Promotes **code reusability and maintainability**.
- Ensures **controlled access** to class variables through public interfaces.