# Encapsulation

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It refers to the bundling of data (attributes) and methods (functions) that operate on the data into a single unit or class. It restricts direct access to some of an object's components, *which* can be crucial for maintaining the integrity of the data. This is typically done by making the data members (fields) private and providing public getter and setter methods to access and update the values of these fields.

**Key Benefits of Encapsulation:**
**Data Hiding:** The internal state of an object is hidden from the outside. Only the object's own methods can directly access and modify its fields.

**Increased Flexibility:** The implementation of a class can be changed without affecting the classes that use it, provided that the public interface remains consistent.

**Reusability**: Encapsulated classes can be easily reused and maintained.

**Ease of Testing:** Encapsulation improves testability by limiting the exposure of implementation details.

**Example of Encapsulation in Java**
Consider a scenario where we need to manage bank accounts. We want to ensure that the balance of a bank account cannot be set to an invalid value directly.

BankAccount Class with Encapsulation:

```
public class BankAccount {
    // Private fields - data encapsulation
    private String accountNumber;
    private double balance;

 // Constructor
 public BankAccount(String accountNumber, double initialBalance)    {
        this.accountNumber = accountNumber;
    setBalance(initialBalance); // Using setter for validation
    }

    // Public getter for accountNumber
```

```
public String getAccountNumber() {
    return accountNumber;
}

// Public getter for balance
public double getBalance() {
    return balance;
}

// Public setter for balance with validation
public void setBalance(double balance) {
    if (balance >= 0) {
        this.balance = balance;
    } else {
        System.out.println("Balance cannot be negative.");
    }
}

// Method to deposit money with encapsulation
public void deposit(double amount) {
    if (amount > 0) {
        balance += amount;
    } else {
     System.out.println("Deposit amount must be positive.");
    }
}

// Method to withdraw money with encapsulation
public void withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
    } else {
        System.out.println("Invalid withdraw amount.");
    }
}

// Method to display account information
```

```
    public void displayAccountInfo() {
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Current Balance: " + balance);
    }
}
```

Using the BankAccount Class:

```
public class Main {
    public static void main(String[] args) {
        // Create a new bank account with initial balance
      BankAccount account = new BankAccount("123456789", 500.0);

      // Display account information
      account.displayAccountInfo();

      // Deposit money
      account.deposit(150.0);
      account.displayAccountInfo();

      // Withdraw money
      account.withdraw(100.0);
      account.displayAccountInfo();

// Try to set an invalid balance directly (will not compile)
// account.balance = -100; // Error: balance has private access in
BankAccount

// Try to set an invalid balance using setter
account.setBalance(-100); // Prints: Balance cannot be negative.

// Try to deposit a negative amount
account.deposit(-50); //Prints: Deposit amount must be positive.

        // Try to withdraw more than the balance
account.withdraw(1000); // Prints: Invalid withdraw amount.
    }
}
```

**Detailed Explanation:**

**Private Fields:**

accountNumber and balance are declared as private. This means they cannot be accessed directly from outside the class.

**Public Getters and Setters:**

getAccountNumber() and getBalance() provide read access to the private fields.
setBalance() allows controlled modification of the balance field. It includes validation to ensure the balance cannot be set to a negative value.

**Public Methods:**

deposit(double amount) and withdraw(double amount) are methods that modify the balance in a controlled manner, ensuring business rules are adhered to (e.g., deposit amount must be positive, withdrawal amount must not exceed the balance).

**Data Hiding and Validation:**

By hiding the fields and providing public methods, we ensure that any changes to the fields are validated, protecting the integrity of the object's state.

**Class Reusability and Maintenance:**

The class can be used and reused without worrying about its internal implementation, which can be modified without affecting other parts of the application as long as the public interface remains the same.

Encapsulation in this way ensures that the BankAccount class maintains a consistent and valid state, prevents invalid operations, and makes the class easier to maintain and extend.