Encapsulation in Java

Encapsulation is a fundamental principle of Object-Oriented Programming (OOP) that bundles the data (variables) and the methods (functions) that operate on the data into a single unit, i.e., a class. It restricts direct access to some of the object's components, allowing data hiding, which ensures better control and integrity of the data.

Encapsulation helps in:

- **Protecting sensitive data** from unauthorized access.
- **Achieving data hiding** by restricting the access to the internals of an object.
- **Providing a controlled interface** to modify the data using getter and setter methods.

Key Components of Encapsulation:

1. **Access Modifiers (private, public, protected)**
2. **Getters and Setters**

---

1. Access Modifiers

Access modifiers define the visibility or scope of a class, variable, method, or constructor. Java provides four main access levels:

Modifier	Class	Package	Subclass	World
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

- **Private:** The `private` access modifier is used to limit the visibility of a member (variable or method) only within the same class. No external class can access or modify this member directly.
- **Public:** The `public` access modifier allows the class members to be accessible from any other class.
- **Protected:** The `protected` modifier allows access to the member within the same package or subclasses in other packages.

Example of Access Modifiers:

```
class BankAccount {
    private double balance; // private variable

    // Public method to display balance
    public void displayBalance() {
```

```
        System.out.println("Balance: " + balance);
    }

    // Protected method, accessible to subclasses
    protected void addInterest(double rate) {
        balance += balance * rate;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.displayBalance();  // Allowed since displayBalance is public
    }
}
```

Output:

```
Balance: 0.0
```

---

2. Getters and Setters

Encapsulation is achieved in Java by making fields `private` and providing `public` getter and setter methods to access and update the values of these fields. These methods ensure data validation and control over access to the private fields.

- **Getters** allow reading the private variables.
- **Setters** allow modifying the values with validation logic if needed.

Example of Getters and Setters:

```
class Employee {
    private String name; // private variable
    private int age;

    // Getter method for 'name'
    public String getName() {
        return name;
    }

    // Setter method for 'name'
    public void setName(String name) {
        this.name = name;
    }

    // Getter method for 'age'
    public int getAge() {
        return age;
```

```
    }

    // Setter method for 'age' with validation
    public void setAge(int age) {
        if(age > 18 && age <= 65) {
            this.age = age;
        } else {
            System.out.println("Invalid age!");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Employee emp = new Employee();

        // Setting values using setters
        emp.setName("John Doe");
        emp.setAge(30);

        // Getting values using getters
        System.out.println("Employee Name: " + emp.getName());
        System.out.println("Employee Age: " + emp.getAge());
    }
}
```

Output:

```
Employee Name: John Doe
Employee Age: 30
```

---

3. Practice: Implement Encapsulation

In this example, we will create a `Student` class with private attributes such as `name` and `grade`. Getters and setters will control access to these fields. The example also demonstrates data validation in the setter.

```
class Student {
    // Private attributes
    private String name;
    private int grade;

    // Constructor
    public Student(String name, int grade) {
        this.name = name;
        setGrade(grade); // Using setter to apply validation
    }

    // Getter for 'name'
```

```
    public String getName() {
        return name;
    }

    // Setter for 'name'
    public void setName(String name) {
        this.name = name;
    }

    // Getter for 'grade'
    public int getGrade() {
        return grade;
    }

    // Setter for 'grade' with validation
    public void setGrade(int grade) {
        if (grade >= 0 && grade <= 100) {
            this.grade = grade;
        } else {
            System.out.println("Invalid grade! Must be between 0 and 100.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating a student object
        Student student = new Student("Alice", 95);

        // Accessing private variables using getters
        System.out.println("Student Name: " + student.getName());
        System.out.println("Student Grade: " + student.getGrade());

        // Modifying variables using setters
        student.setGrade(105);  // Invalid grade
        student.setGrade(85);   // Valid grade

        System.out.println("Updated Grade: " + student.getGrade());
    }
}
```

Output:

```
Student Name: Alice
Student Grade: 95
Invalid grade! Must be between 0 and 100.
Updated Grade: 85
```

Key Points of Encapsulation:

- Encapsulation ensures that sensitive data is hidden from outside classes.
- Private fields can only be accessed through public getter and setter methods.
- It provides controlled access to the attributes by including validation logic in the setter methods.

This concept is crucial in real-world applications to maintain data security, integrity, and proper access control.

Let's dive deeper into the concept of **Encapsulation** with more examples and use cases to give you a clearer understanding. I'll add some additional content around **encapsulation in real-world applications**, **common mistakes**, and **best practices**.

4. Why Encapsulation is Important?

Encapsulation provides a mechanism to bundle the data and the methods that manipulate that data into one unit, typically a class. This helps in maintaining **data integrity** and provides several key advantages:

1. **Data Hiding:** Private fields cannot be accessed directly from outside the class, so the data is protected from unintended or malicious modification.
2. **Control over Data:** By using getter and setter methods, you can control what values are assigned to the fields. You can also perform data validation inside the setter methods.
3. **Flexibility and Maintainability:** By encapsulating the data, you can modify the implementation later without affecting the external code that uses the class. For example, you could change the way a field is stored (e.g., from a `String` to an `Array`) without changing how it's accessed.
4. **Loose Coupling:** Encapsulation allows you to change the internals of a class without affecting the external code. This results in looser coupling, which is key for maintaining and scaling codebases.

---

5. Example: Encapsulation in a Real-World Application

Let’s consider a **Banking Application** where you need to manage customer accounts. You don’t want external classes to access and modify the account balance directly. Instead, you provide controlled access to deposit and withdraw money.

Example:

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

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

    // Getter for account number
    public String getAccountNumber() {
        return accountNumber;
```

```
    }

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

    // Method to deposit money with validation
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Invalid deposit amount.");
        }
    }

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

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount("123456789", 500.0);

        // Accessing account balance via getter
        System.out.println("Initial Balance: " + account.getBalance());

        // Depositing money
        account.deposit(200);
        System.out.println("Updated Balance: " + account.getBalance());

        // Trying an invalid deposit
        account.deposit(-50); // Invalid

        // Withdrawing money
        account.withdraw(100);
        System.out.println("Final Balance: " + account.getBalance());
    }
}
```

Output:

```
Initial Balance: 500.0
Deposited: 200.0
```

```
Updated Balance: 700.0
Invalid deposit amount.
Withdrew: 100.0
Final Balance: 600.0
```

In this example:

- The balance is **private**, so no other class can modify it directly.
- Access to the balance is provided through controlled methods like `deposit()` and `withdraw()`, ensuring only valid operations can be performed.

---

6. Common Mistakes in Encapsulation

1. **Exposing Internal Variables:** A common mistake is making fields `public`, which breaks the concept of encapsulation because any external class can directly modify the fields, risking the integrity of the data.
2. **No Validation in Setters:** Not performing proper validation inside setter methods can lead to inconsistent or invalid data being stored. Always include checks when modifying critical data.

```
public void setBalance(double balance) {
    if (balance >= 0) {
        this.balance = balance;
    } else {
        System.out.println("Balance cannot be negative.");
    }
}
```

3. **No Getters/Setters for Sensitive Data:** For highly sensitive data, even getters should be avoided if you do not want to expose that data. For example, password fields in a class should not have public getters.

---

7. Best Practices for Encapsulation

- **Always Use Private Fields:** Make all fields `private` to prevent direct access and modification. Use getters and setters for controlled access.
- **Validation Logic in Setters:** Perform data validation within setters to ensure that only valid values are assigned to fields.
- **Immutability:** If the object's state should not change after creation, you can make the fields `final` and omit the setters to achieve **immutability**. For example:

```
class ImmutableStudent {
    private final String name;
    private final int grade;

    public ImmutableStudent(String name, int grade) {
        this.name = name;
```

```
        this.grade = grade;
    }

    // Only getters, no setters
    public String getName() {
        return name;
    }

    public int getGrade() {
        return grade;
    }
}
```

- **Consistency in Naming Conventions:** Follow standard naming conventions for getters and setters. The getter method for a variable `name` should be `getName()` and the setter should be `setName()`.
- **Use of Constructors:** When initializing fields, constructors are a great way to set values at the time of object creation, allowing you to create an object in a consistent state.

---

8. Advanced Encapsulation: Accessing Encapsulated Data Through Methods

In some cases, you might want to provide calculated or derived data without directly exposing the internal state. This can also be done through encapsulation.

Example: Calculating Interest in a Bank Account

```
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
    }

    public double getBalance() {
        return balance;
    }

    // Method to calculate interest without modifying balance directly
    public double calculateInterest(double interestRate) {
        return balance * interestRate;
    }
}

public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000.0);
        System.out.println("Balance: " + account.getBalance());
        double interest = account.calculateInterest(0.05);
        System.out.println("Interest at 5%: " + interest);
```

```
    }
}
```

Output:

```
Balance: 1000.0
Interest at 5%: 50.0
```

Here, the internal state (`balance`) is still private, but we can calculate derived data (`interest`) using a method without exposing the raw data directly.

Conclusion

Encapsulation is a powerful concept that ensures better control over the internal workings of a class. By restricting direct access to class fields and using getters and setters, you can enforce data validation and integrity. This leads to more maintainable, flexible, and secure code.