Encapsulation

Encapsulation is one of the four foundational principles of object-oriented design. It is the practice of grouping data (variables) and behavior (methods) that operate on that data into a single unit (typically a class) and restricting direct access to the internal details of that class.

In simple terms:

Encapsulation = Data hiding + Controlled access

Real-World Analogy
Think of a bank account as a vault inside a bank. You don't get to walk into the vault and change the numbers yourself.

Instead, you interact through an ATM interface.
The ATM offers specific operations: deposit, withdraw, check balance.
You can’t see or touch the actual ledger that tracks your money.
Internally, the bank may change how it calculates interest or stores data, but your interaction remains the same.
In programming, Encapsulation is typically achieved using:

Access Modifiers (private, protected, public)
Getter and Setters
Why Encapsulation Matters
It helps you build systems that are robust, secure, and easy to maintain. Here's why it's essential:

Data Hiding: By restricting direct access to internal fields, encapsulation shields sensitive data from unintended interference or misuse.
Controlled Access and Security: : It ensures that data can only be accessed or modified through well-defined methods, allowing you to enforce rules, validation, or logging when needed.
Improved Maintainability: Because internal implementation details are hidden, you can refactor or optimize them without affecting the external code that depends on the class.
In a well-encapsulated design, external code doesn’t need to know how something is done, it only needs to know what can be done.

Code Example
Let’s look at two practical examples that demonstrate how encapsulation helps you protect internal state, enforce business rules, and expose only what’s necessary.

Example 1: BankAccount
In a banking system, you want users to deposit and withdraw funds, but you must prevent direct manipulation of the account balance. Here’s how encapsulation helps:

Java

```
public class BankAccount {
    private double balance; // Internal state is hidden

    public void deposit(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Deposit amount must be positive");
        }
        balance += amount;
    }

    public void withdraw(double amount) {
        if (amount <= 0) {
            throw new IllegalArgumentException("Withdrawal amount must be positive");
        }
        if (amount > balance) {
            throw new IllegalArgumentException("Insufficient funds");
        }
        balance -= amount;
    }

    public double getBalance() {
        return balance; // Controlled access
    }
}
```

Python

```
class BankAccount:
    def __init__(self):
        self.__balance = 0.0  # Internal state is hidden

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > self.__balance:
            raise ValueError("Insufficient funds")
        self.__balance -= amount
```

```
def get_balance(self):
    return self.__balance  # Controlled access
```

- `balance` is marked `private`, so no external class can access or modify it directly.
- `deposit()` and `withdraw()` are **public entry points** that validate user input before updating the state.
- `getBalance()` allows **read-only access**, without revealing the underlying variable or letting external code change it.

This ensures the account remains in a valid state at all times and business rules are enforced through controlled interfaces.

Example 2: PaymentProcessor

In this example, a `PaymentProcessor` class accepts a card number and amount, but internally **masks the card number** to protect user privacy. Again, encapsulation allows us to hide implementation details while offering a clean interface.

Python

```
class PaymentProcessor:
    def __init__(self, card_number, amount):
        self.__card_number = self.__mask_card_number(card_number)
        self.__amount = amount
    def __mask_card_number(self, card_number):
        return "****-****-****-" + card_number[-4:]

def process_payment(self):
        print(f"Processing payment of {self.__amount} for card {self.__card_number}")

if __name__ == "__main__":
    payment = PaymentProcessor("1234567812345678", 250.00)
    payment.process_payment()
```

Output:

```
Processing payment of 250.0 for card ****-****-****-5678
```

Java

```
class PaymentProcessor {
```

```
    private String cardNumber;
    private double amount;

    public PaymentProcessor(String cardNumber, double amount) {
        this.cardNumber = maskCardNumber(cardNumber);
        this.amount = amount;
    }

    private String maskCardNumber(String cardNumber) {
        return "****-****-****-" +
cardNumber.substring(cardNumber.length() - 4);
    }

    public void processPayment() {
        System.out.println("Processing payment of " + amount + "
for card " + cardNumber);
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentProcessor payment = new
PaymentProcessor("1234567812345678", 250.00);
        payment.processPayment();
    }
}
```

- The raw card number is never stored or exposed directly.
- Masking is handled **internally** via a private method.
- The external caller doesn't need to know how masking is done, they just call `processPayment()`.

This design secures sensitive data and centralizes masking logic in one place, making the system safer and easier to maintain.