객체지향 프로그래밍 (전반부)

객체지향 프로그래밍

- Object-Oriented Programming(OOP)
- 객체들의 집합으로 프로그램의 상호 작용을 표현
- 데이터를 객체로 취급하여 객체 내부에 선언된 메서드를 활용하는 방식
- 설계에 많은 시간이 소요되며, 처리 속도가 다른 프로그래밍 패러다임에 비해 상대적으로 느림

```javascript
// JavaScript
const lst = [1, 2, 3, 4, 5, 11, 12]
class List {
    constructor(list) {
        this.list = list
        this.mx = list.reduce((max, num) => num > max ? num : max, 0)
    }
    getMax() {
        return this.mx
    }
}
const a = new List(lst)
console.log(a.getMax())    // 12
```

```java
// Java
import java.util.Arrays;

class List {
    private int[] list;
    private int mx;

    public List(int[] list) {
        this.list = list;
        this.mx = Arrays.stream(list).max().getAsInt();
    }

    public int getMax() {
        return this.mx;
    }
}

public class Main {
    public static void main(String[] args) {
        List a = new List(new int[]{1, 2, 3, 4, 5, 11, 12});
        System.out.println(a.getMax());  // 12
    }
}
```

```python
# Python
class List:
    def __init__(self, list):
        self.list = list
        self.mx = max(list)

    def getMax(self):
```

```
        return self.mx

lst = [1, 2, 3, 4, 5, 11, 12]
a = List(lst)
print(a.getMax())  # 12
```

객체지향 프로그래밍의 특징

추상화(abstraction)

- 실생활의 복잡한 시스템으로부터 핵심적인 개념 또는 기능을 간추려내는 것

추상화의 핵심 요소

1. 핵심 개념 도출
    - **실**세계의 복잡한 객체나 개념에서 가장 중요한 속성과 행위만을 추려내어 정의함
    - e.g.) '자동차'라는 객체를 모델링할 때 '색상', '브랜드', '속도' 등의 속성과 '가속하기', '멈추기' 같은 행위를 추상화하여 클래스에 포함시킬 수 있음
2. 공통성의 추출
    - 서로 다른 객체들 사이의 공통적인 속성과 행위를 찾아내어 공통의 추상 클래스나 인터페이스를 만듦
    - e.g.) '자동차', '트럭', '오토바이'는 모두 '탈 것'이라는 더 큰 카테고리에 속할 수 있으며, 이들은 '이동하기', '멈추기' 등의 공통 행위를 가지고 있을 수 있음

추상화를 사용하는 이유

- 복잡성 관리
    - 실세계의 복잡한 시스템을 간단하고 이해하기 쉬운 모델로 변환하여, 개발자가 더 쉽게 다룰 수 있게 함
- 재사용성 향상
    - 공통적인 속성과 행위를 추상 클래스나 인터페이스로 정의함으로써, 코드의 재사용성을 높이고 유지보수를 용이하게 함
- 확장성
    - 추상화를 통해 정의된 클래스는 새로운 클래스를 쉽게 추가하거나 기존 클래스를 확장하여 시스템을 발전시킬 수 있음
    - 이는 개방-폐쇄 원칙(Open/Closed Principle)을 지원하여, 기존 코드를 변경하지 않고도 시스템의 기능을 확장할 수 있게 합니다.

캡슐화(encapsulation)

- 객체의 속성과 메소드를 하나로 묶고 일부를 외부에 감추어 은닉하는 것
- 객체의 세부 구현 내용을 숨기고, 사용자에게는 필요한 필드와 메소드만을 노출시켜 객체의 데이터와 행동을 하나로 묶는 기법

```
public class BankAccount {
    private double balance; // 외부에서 직접 접근할 수 없음

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

    // 입금
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    // 출금
    public void withdraw(double amount) {
```

```
        if (amount > 0 && balance >= amount) {
            balance -= amount;
        }
    }

    // 잔액 조회
    public double getBalance() {
        return balance;
    }
}
```

캡슐화의 핵심 요소

1. 정보 은닉(Information Hiding)
    - 객체의 상태를 나타내는 데이터(속성)와 그 데이터를 조작할 수 있는 메소드(행동)를 하나로 묶고, 객체 외부에서는 직접적으로 접근할 수 없도록 제한
    - 객체가 제공하는 메서드를 통해서만 내부 상태를 변경할 수 있게 함으로써 데이터의 무결성을 유지
2. 인터페이스 제공
    - 외부에는 객체와 상호작용하기 위한 인터페이스를 제공하면서, 객체의 내부 구현 세부 사항은 숨김
    - 이 인터페이스를 통해 사용자는 객체의 구현 방법을 몰라도 해당 객체의 기능을 사용할 수 있음

캡슐화를 사용하는 이유

- 데이터 보호
    - 객체의 중요한 데이터를 외부의 무분별한 접근으로부터 보호하여 데이터의 무결성과 안정성을 보장
- 유지보수성 향상
    - 객체의 내부 구현이 외부에 노출되지 않기 때문에, 내부 구현을 변경하더라도 외부에 영향을 미치지 않아 유지보수가 용이함
- 사용 편의성
    - 사용자는 객체가 어떻게 구현되었는지 알 필요 없이, 제공된 인터페이스를 통해 객체를 쉽게 사용할 수 있음

상속성(inheritance)

- 상위 클래스의 필드와 메소드를 하위 클래스가 이어받아서 재사용하거나 추가, 확장하는 것
- 코드의 재사용 측면, 계층적인 관계 생성, 유지 보수성 측면에서 중요함

다형성(polymorphism)

- 하나의 메소드나 클래스가 다양한 방법으로 동작하는 것
- 대표적으로 오버로딩, 오버라이딩이 있음

1️⃣ 오버로딩(overloading)

- 같은 이름을 가진 메소드를 여러 개 두는 것
- 메소드의 타입, 매개변수의 유형, 개수 등으로 구별하여 여러 개를 둘 수 있음
- 컴파일 중에 발생하는 '정적' 다형성

```
class Person {

    public void eat(String a) {
        System.out.println("I eat " + a);
    }

    public void eat(String a, String b) {
        System.out.println("I eat " + a + " and " + b);
```

```
    }
}

public class CalculateArea {

    public static void main(String[] args) {
        Person a = new Person();
        a.eat("apple");
        a.eat("tomato", "phodo");
    }
}
```

2️⃣ 오버라이딩(overriding)

- 주로 메소드 오버라이딩(method overriding)을 말함
- 상위 클래스로부터 상속받은 메소드를 하위 클래스가 제정의 하는 것을 의미함

```
class Animal {
    public void bark() {
        System.out.println("mumu! mumu!");
    }
}

class Dog extends Animal {
    @Override
    public void bark() {
        System.out.println("wal!!! wal!!!");
    }
}

public class Main {

    public static void main(String[] args) {
        Dog d = new Dog();
        d.bark();
    }
}
```