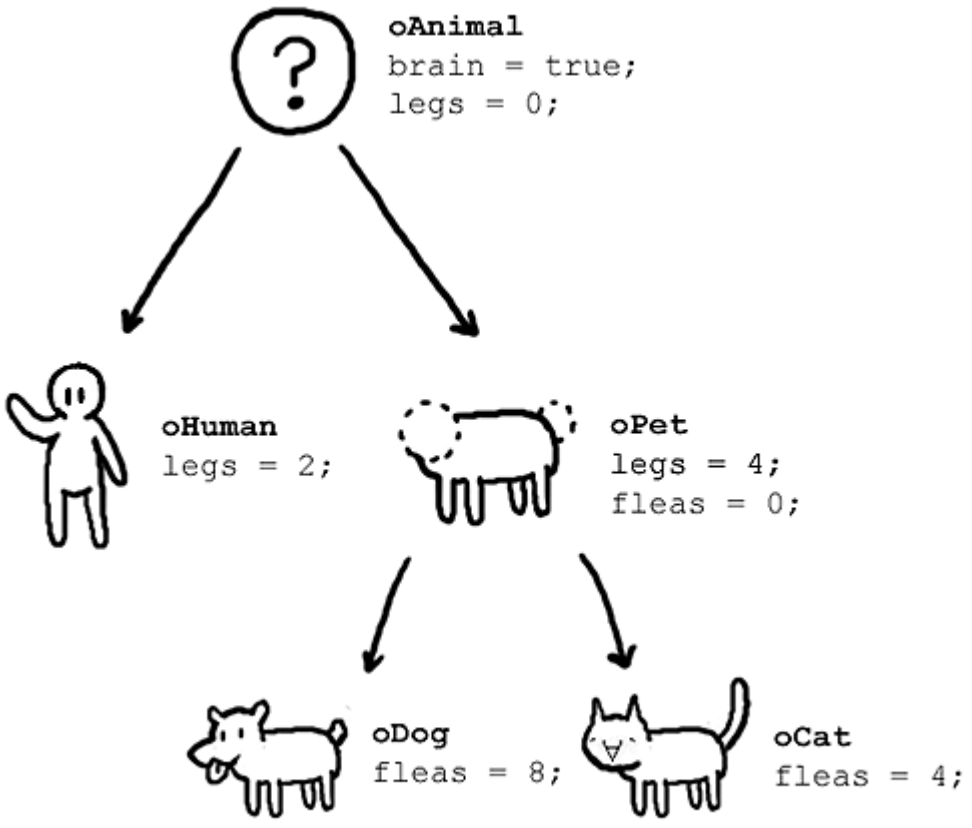
객체 지향 프로그래밍

객체 지향 프로그래밍(Oject-Oriented Programming, OOP)

객체 지향 프로그래밍이란 프로그래밍에서 필요한 **데이터를 추상화 시켜 상태와 행위를 가진 객체로 만들고, 객체들간의 상호작용을 통해 로직을 구성하는 프로그래밍 방법**이다.



객체

객체는 프로그램에서 사용되는 **데이터 또는 식별자에 의해 참조되는 공간**을 의미하며 **값을 저장할 변수와 작업을 수행할 메서드를 서로 연관된 것들끼리 묶어 만든 것**이다. 객체 지향 프로그래밍을 레고에 빗대 표현하기도 하는데, 객체가 레고의 조각인 것이고 객체를 이용해 무언가 만드는 방식을 객체지향 프로그래밍이라고도 비유한다. 객체는 레고 조각으로 비유했기 때문에 **여러 군데에서 재사용이 가능**하고 이는 **부품화**와 **재사용성** 이라는 OOP의 특징을 보여준다.

객체 지향 프로그래밍을 지원하는 언어로는 C++, C#, Java, Python, JavaScript 등이 있다.

특징

객체 지향 프로그래밍은 크게 **추상화, 캡슐화, 상속, 다형성**의 4가지 특징을 가진다.

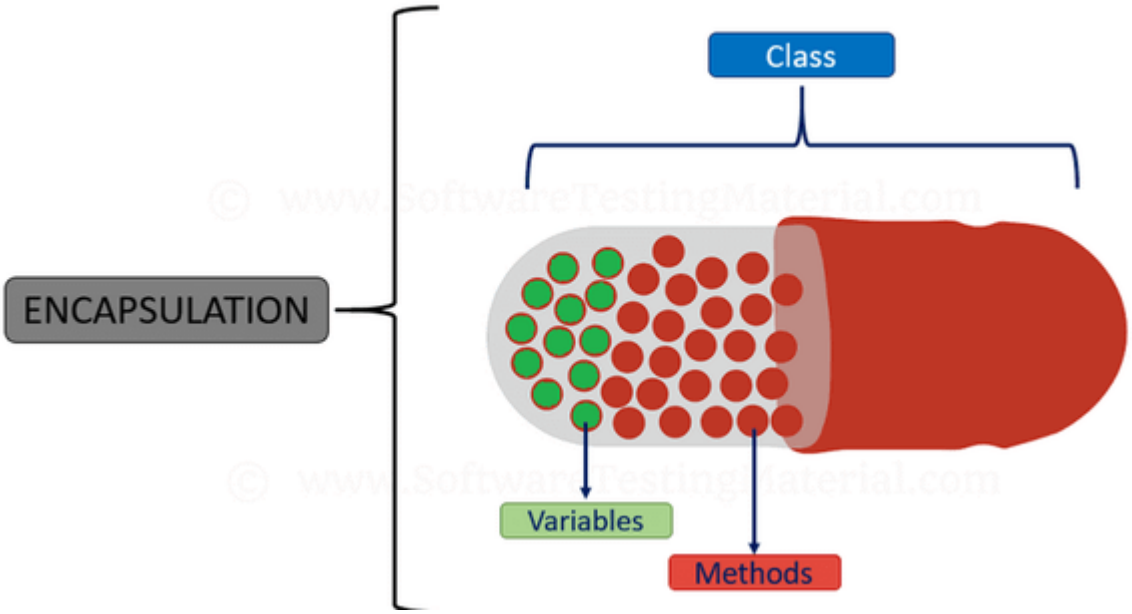
추상화

객체에서 공통된 속성과 행위를 추출하는 것으로 공통의 속성과 행위를 차장 타입을 정의하는 과정이다. 추상화는 **불필요한 정보는 숨기고 중요한 정보만을 표현**함으로써 **프로그램을 간단하게 만드는 역할**을 한다.

아우디, 니싼, 볼보 모두 '자동차'에 해당되기 때문에 자동차라는 추상화 집합을 만들어두고 자동차들이 가진 특징들을 만들어 활용한다. 이 과정에서 '기아' 와 같은 다른 자동차 브랜드가 추가되더라도, '자동차' 가 구현되어 있다면 공통된 부분에 대한 코드 작업이 필요하지 않다.

캡슐화

캡슐화란 쉽게 말해 **변수나 메서드 들을 캡슐로 감싸 안보이게 하는 정보 은닉 개념 중 하나**이다. 목적은 **객체가 기능이나 변수를 어떻게 구현했는지 외부에 감추는 것**이다.



자바에서는 대표적으로 protected, default, private 의 접근 제어자를 통해 구현이 가능하다.

```
class Time {
    private int hour; // hour는 외부에서 접근하지 못하게 private으로

    // Setter
    public void setHour(int hour) {
        if (hour < 0 || hour > 24) { // hour에 대한 유효성 검사
            return;
        } else {
            this.hour = hour;
        }
    }

    // Getter
    public int getHour() {
        return hour;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Time time = new Time();

        // 유효하지 않은 parameter 입력
        time.setHour(25);
        System.out.println("Hour: " + time.getHour()); // 0

        // 유효한 parameter 입력
        time.setHour(13);
        System.out.println("Hour: " + time.getHour()); // 13
    }
}
```

위와 같이 코드를 작성하면 `private hour` 라는 변수를 다루기 위해서는 setter 메서드를 사용해야 하고, 호출을 위해선 getter 메서드를 호출해야한다. 이렇게 getter/setter 를 이용하는 방식이 좋은 점은 유효하지 않은 숫자로 직접 변수 `hour` 를 세팅하려는 시도를 메서드 내부에서 유효성 체크 로직을 통해 거를 수 있다는 점이다.

상속화

클래스의 속성과 행위를 하위 클래스에 물려주거나 하위 클래스가 상위 클래스의 속성 및 행위를 물려받는 것을 말합니다. 새로운 클래스가 기존 클래스의 데이터와 연산을 이용할 수 있게 하는 기능으로 다음과 같은 장단점이 있습니다.

1. **장점**
    - 재사용으로 인한 코드가 줄어든다.
    - 범용적인 사용이 가능하다.
    - 자료와 메서드의 자유로운 사용 및 추가가 가능하다.

2. **단점**

- 상위 클래스의 변경이 어려워진다.
- 불필요한 클래스가 증가할 수 있다.
- 상속이 잘못 사용될 수 있다.

다형성

하나의 변수명, 함수명이 상황에 따라 다른 의미로 해석 될 수 있는 것을 의미하며, 어떠한 요소에 여러 개념을 넣어놓는 것입니다.

OOP는 하나의 클래스 내부에 같은 이름의 행위를 여러 개 정의하거나, 상위 클래스의 행위를 하위 클래스에서 재정의해 사용할 수 있기 때문에 이런 특징을 갖게 되었습니다.

오버라이딩

- 상위 클래스가 가지고 있는 메서드를 하위 클래스가 재정의해 사용하는 것

오버로딩

- 같은 이름의 메서드가 인자의 개수나 자료형에 따라 다른 기능을 하는 것

장단점

장점

- 클래스 단위로 모듈화시켜 개발하기 때문에 업무 분담이 편리하고 대규모 소프트웨어 개발에 적합하다.
- 클래스 단위로 수정이 가능하기 때문에 유지보수가 편리하다.
- 클래스를 재사용하거나 상속을 통해 확장함으로써 코드 재사용이 용이하다.

단점

- 처리속도가 상대적으로 느리다.
- 객체의 수가 많아짐에 따라 용량이 커질 수 있다.
- 설계 시 많은 시간과 노력이 필요하게 될 수 있다.

SOLID (객체 지향 설계 원칙)

객체 지향적으로 설계하기 위해서는 SOLID라 불리는 5가지 원칙을 지켜야 한다.

1. **단일 책임 원칙 (SRP)**
    - 하나의 클래스는 단 하나의 책임만 가져야 한다.
    - SRP를 지키지 않을 경우 한 책임의 변경에 의해 다른 책임과 관련된 코드에 영향이 갈 수 있다.

2. **개방-폐쇄 원칙 (OCP)**
    - 소프트웨어 요소는 확장에는 열려 있으나 변경에는 닫혀 있어야 한다.
    - 기능을 변경하거나 확장할 수 있으면서 기능을 사용하는 코드는 수정하지 않는다.

3. **리스코프 치환 원칙 (LSP)**
    - 프로그램 객체는 프로그램의 정확성을 깨트리지 않으면서 하위 타입의 인스턴스로 바꿀 수 있어야 한다.
    - 상위 타입의 객체를 하위 타입의 객체로 치환해도, 상위 타입을 사용하는 프로그램은 정상적으로 동작해야 한다.

4. **인터페이스 분리 원칙 (ISP)**
    - 범용 인터페이스 하나보다 클라이언트를 위한 여러 개의 인터페이스로 구성하는 것이 좋다.

- 인터페이스는 인터페이스를 사용하는 클라이언트를 기준으로 분리해야 한다.
- 클라이언트가 필요로 하는 인터페이스로 분리함으로써 각 클라이언트가 사용하지 않는 인터페이스에 변경이 있어도 영향을 받지 않도록 만들어야 한다.

5. **의존 관계 역전 원칙 (DIP)**
    - 추상화에 의존해야지 구체화에 의존하면 안된다
    - 고수준 모듈은 저수준 모듈의 구현에 의존해서는 안되고 저수준 모듈은 고수준 모듈에서 정의한 추상 타입에 의존해야 한다.