OOP at JavaScript and TypeScript

내가 생각하는 객체지향 프로그래밍

객체를 독립된 주체로서 이해하는가?

객체를 하나의 데이터와 기능을 가진 물리적인 존재로 이해한다.

클래스와 객체의 관계는 붕어빵틀과 붕어빵이 아니다.

```
class 붕어빵틀(price, name) {
    ...
}

let 붕어빵 = new 붕어빵틀(1000, "Jin");
```

- 사람은 클래스인가? 객체인가?
- 김연아는 클래스인가? 객체인가?
- 뽀로로는 클래스인가? 객체인가?
- 펭귄은 클래스인가? 객체인가?

클래스는 객체를 만들어내는 생성도이다.

클래스는 같은 특성을 지닌 여러 객체를 총칭하는 집합의 개념이다.

```
클래스 : 객체 = 사람 : 김연아 = 쥐 : 미키마우스

사람 홍길동 = new 사람(); // 인스턴스 = new 클래스();
사람 줄리엣 = new 사람(); // 인스턴스가 객체고 클래스는 객체의 설계도이다.
```

객체지향 핵심 개념

- Class
- 캡슐화(Encapsulation)
- 상속(Inheritance)
- 다형성(polymorphism)
- 추상화(abstraction)

초등학생에게 설명해보자.


아빠 캡슐화가 무야
2
오후 7:13
2
캡슐하면 떠오르는게 뭐야?
오후 7:14
아 들 래 미
뭔가를 감싸는거?
2
오후 7:14
2
그치. 그럼 뭘 감쌀까?
오후 7:15
아 들 래 미
음 객체?
2
객체에 담긴 정보?
2
오후 7:15
2
정보만 있나?
오후 7:15
아 들 래 미
음 메소드?
2
오후 7:15

클래스 (Class)

JavaScript

- 함수가 곧 클래스의 역할을 자임한다.
- 객체지향 언어에서는 객체의 클래스를 정의한 다음, 해당 클래스의 인스턴스를 각각 생성한다.
- 자바스크립트에서는 객체를 생성하기 위한 생성자 함수를 지정할 수 있다. 프로퍼티가 없는 새로운 객체를 생성한 후, 생성자 함수를 호출하고 새 객체를 `this` 키워드와 함께 전달한다.

- 일반적으로 생성자 함수는 반환값이 없다.
- `this` 프로퍼티를 추가하는 함수를 작성한다.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

point1 = new Point(5, 5);
point2 = new Point(8, 8);

console.log(JSON.stringify(point1));
console.log(JSON.stringify(point2));
```

- 모든 자바스크립트 객체는 프로토타입 객체에 대한 내부 참조를 갖고 있다. 프로토타입 객체로부터 프로퍼티를 상속받는다.
- 모든 함수는 함수가 정의됐을 때 자동으로 생성되고 초기화되는 `prototype` 프로퍼티를 가지고 있다. 메서드와 프로퍼티는 객체가 생성된 이후에도 동적으로 프로토타입에 추가될 수 있다.

```
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.add = function () {
  return this.x + this.y;
};

point1 = new Point(5, 5);
point2 = new Point(8, 8);

console.log(point1.add());
console.log(point2.add());
```

프로토타입을 왜 사용하는가?

- 상속이 가능하므로 객체 중심적으로 프로그래밍할 수 있다.
- 객체가 프로토타입의 프로퍼티를 상속하므로 객체가 필요한 메모리 양을 줄일 수 있다.
- 만약 Point 객체에서 프로토타입에서 프로퍼티를 선언하지 않고 생성자 함수로 생성하고자 한다면 인스턴스 별로 각각의 변수가 서로 다른 메모리 주소를 가질 것이다.

ES6의 클래스 키워드는 무엇인가?

- 자바스크립트에서 1급 시민은 함수이다. 따라서 클래스 키워드는 자바스크립트에서 클래스를 더 잘 표현하기 위한 수단 중 하나로, 또 하나의 함수 표현식일 뿐이다. (ex: getter, setter)

TypeScript

- ES6에서 불가능한 다양한 기능을 제공한다.
- 접근 제한자를 Java처럼 다양하게 가져갈 수 있다.
- static 속성을 메소드와 프로퍼티 모두에서 적용할 수 있다.

```
class Person {
    private name: string
    protected pName: string
    constructor(name: string) {
        this.name = name
        this.pName = name
    }
    static greet(): string {
        return 'Hello, ' + "soo"
    }
}
```

캡슐화 (Encapsulation)

- 객체의 기능과 정보를 어떻게 구현하는 지 숨겨야 한다.
- 구현에 사용된 상세 로직과 정보를 외부에 숨긴다.
- 외부 조건에 영향을 받지 않고 객체 내부의 구현을 변경할 수 있다.
- 외부 소스코드에 따라 연쇄적인 반응을 겪지 않으므로, 리팩토링이 쉽고 테스트에 용이하다.
- 초등학생에게 설명하기

```
public class Car {
    public String carName;
    public int move;

    public Car(String carName, int move) {
        this.carName = carName;
        this.move = move;
    }

    public void move(int number) {
        if (number >= FORWARD_NUM) {
            position++;
        }
    }
}

Car car = new Car();
car.move(new Random().nextInt(10));
```

Module Pattern

```
let Developer = function(arg) {
  let lang = arg ? arg : '';

  return {
    getLang : function() {
      return lang;
    },
    setLang : function(arg) {
      lang = arg;
    }

  }
};

let bkjang = new Developer('javascript');

console.log(bkjang.getLang()); //javascript

bkjang.setLang('java');

console.log(bkjang.getLang()); //java
```

- Developer 생성자 함수에서 `this`가 아닌 `let lang = arg ? arg : '';`으로 선언하면 자바스크립트는 함수형 스코프를 따르기 때문에 private해진다.
- `getLang()` 과 `setLang()` 이라는 함수는 클로저이기 때문에 외부에서는 lang이라는 변수의 값에 접근할 수 있는 인터페이스가 된다.
- 위와 같이 `getLang()`과 `setLang()`과 같은 public 메서드를 인터페이스로 제공하고 lang과 같은 private한 변수에 인터페이스를 통해서만 접근하도록 하는 것이 모듈 패턴이다.

```
var Developer = function (obj) {
  var developerInfo = obj;

  return {
    getDeveloperInfo: function() {
      return developerInfo;
    }
  };
};

var developer = new Developer({ name: 'BKJang', lang: 'javascript' });

var bkJang = developer.getDeveloperInfo();
console.log('bkJang: ', bkJang);
// bkJang:  {name: "BKJang", lang: "javascript"}

bkJang.lang = 'java'; //인터페이스가 아닌 직접 변경

bkJang = developer.getDeveloperInfo();
console.log('bkJang: ', bkJang);
// bkJang:  {name: "BKJang", lang: "java"}

console.log(Developer.prototype === bkJang.__proto__); //false
```

- 일반 변수가 아닌 객체나 배열을 멤버 변수로 가지고 이를 그대로 반환할 경우, 외부에서 이 멤버를 변경할 수 있다.
- 왜냐하면, 객체나 배열을 반환하는 경우는 얕은 복사(shallow copy)로 private 멤버의 참조값을 반환하게 된다.
- 따라서, 반환할 객체나 배열의 정보를 담은 새로운 객체를 만들어 깊은 복사(deep copy)를 거친 후 반환해야 한다.
- 또한, 위처럼 일반 객체를 반환하면 프로토타입 객체는 Object.prototype 객체가 되기 때문에 상속을 구현할 수 없다. 따라서 함수를 반환해야 한다.

```
var Developer = (function() {
  var lang;

  //생성자 정의
  function Developer(arg) {
    lang = arg ? arg : '';
  }

  Developer.prototype = {
    getLang : function() {
      return lang;
    },
    setLang : function(arg) {
      lang = arg;
    }
  }

  return Developer;
}());

var bkJang = new Developer('javscript');

console.log(bkJang.getLang()); //javscript

bkJang.lang = 'java'; //인터페이스를 통해서가 아닌 직접 변경
console.log(bkJang.getLang()); //javscript

bkJang.setLang('java');
console.log(bkJang.getLang()); //java

console.log(Developer.prototype === bkJang.__proto__); //true
```

- 마지막 출력 값을 보면 인스턴스인 bkJang의 프로토타입 객체가 Developer.prototype 객체임을 알 수 있고 이는 상속을 구현할 수 있음을 의미한다.
- 참고 링크

TypeScript

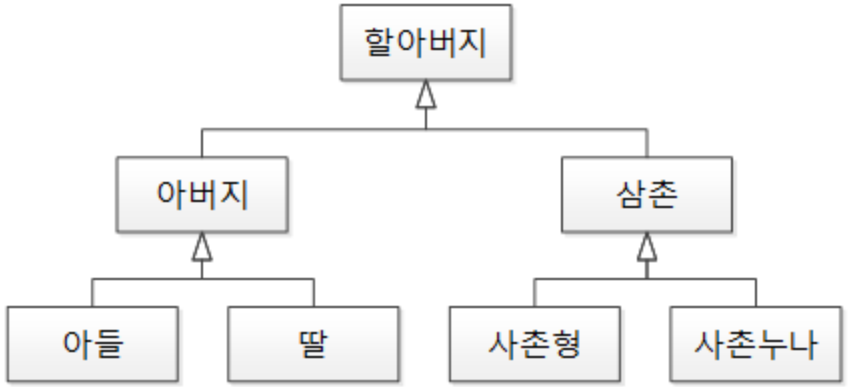
```
class Totalizer {
    private total = 0;
    private taxRateFactor = 0.2;

    addDonation(amount: number) {
        if (amount <= 0) {
            throw new Error('Donation exception');
        }
        var taxRebate = amount * this.taxRateFactor;
        var totalDonation = amount + taxRebate;
        this.total += totalDonation;
    }

    getAmountRaised() {
        return this.total;
    }
}

var totalizer = new Totalizer();
totalizer.addDonation(100.00);

var fundsRaised = totalizer.getAmountRaised();

// 120
console.log(fundsRaised);
```

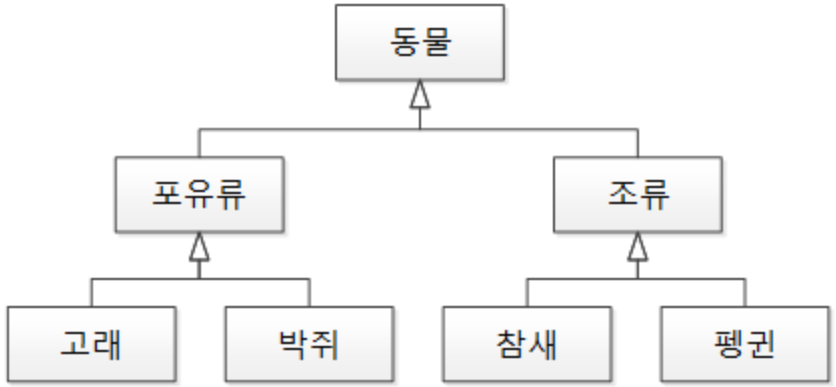
상속 (Inheritance)

확장 (Extend)

> "서브 타입은 언제나 자신의 기반 타입으로 교체할 수 있어야 한다." - 로버트 C 마틴

- 객체지향에서 상속은 조직도, 계층도가 아니라 분류도가 되어야 한다.
- 하위 클래스 is a kind of 상위 클래스 : 하위 분류는 상위 분류의 한 종류이다.
- 구현 클래스 is able to 인터페이스 : 구현 분류는 인터페이스 할 수 있어야한다.

- LSP 위반 사례 : 딸은 아버지의 역할을 할 수 없다.
- 

- LSP를 잘 구현한 사례 : 박쥐는 포유류의 역할을 할 수 있다.
- 

- Person 프로토타입 멤버의 상속이란 결국 프로토타입 체인상에서 Korean의 프로토타입 객체의 상위에 Person의 프로토타입 객체를 만들어 끼워넣는 작업이다.

```
function Person() {}
Person.prototype.species = "human";

function Korean() {}
Korean.prototype.nationality = "korea";

Korean.prototype = new Person();
// Korean 프로토타입 객체로 Person의 인스턴스를 사용
Korean.prototype.constructor = Korean;
// Korean의 프로토타입 객체의 constructor 속성값을 Korean 생성자에 대한 참조

let korean = new Korean();
let species = korean.species;
console.log(species); // human 반환
let nationality = korean.nationality;
console.log(nationality); // korean 반환
```

다형성

- 인터페이스와 추상 클래스

```
// 인터페이스
function Runable() {
        // 추상메서드
        this.run = function() {
            throw new Error("run메서드 미구현");
        }
}
// 구현클래스
// RunImpl에서 run메서드를 구현하지 않으면 부모클래스에서 에러를 던진다. (구현 강제)
function RunImpl() {
        // Override
        this.run = function() {
            console.log("달려!!!!");
        }
}
RunImpl.prototype = new Runable();  // 상속
var man = new RunImpl();  // 객체생성
// 부모클래스에 같은 이름의 메서드가 있어도 부모클래스를 상속받은 구현클래스의 재정의된 메서드가 실행된다.
// (Scope chain(스코프체인)에서 가장 먼저 발견된 메서드를 실행하기 때문이다)
man.run();

// 다형성 체크
// instanceof : 개체가 특정 클래스의 인스턴스인지 여부
console.log(man instanceof Object);    // true
console.log(man instanceof Runable);  // true
console.log(man instanceof RunImpl);  // true
```

```
// Animal.js
class Animal {
    constructor(name) {
        this.name = name;
    }

    getName() {
        return this.name;
    }

    // Abstract
    makeNoise() {
        throw new Error('makeNoise() must be implement.');
    }
}

export default Animal;
```