클래스와 객체

ES6부터 Javascript도 Java와 같은 Class 기반의 객체지향을 지원하기 시작했습니다. ES5시절에는 Prototype 기반 객체지향을 사용했지만 문법이 워낙 특이하기 때문에 접근하기가 쉽지 않았지만 이제는 다른 프로그래밍 언어들과 비슷한 분법으로 객체지향의 구현이 가능해 졌습니다. Class와 객체의 개념은 리액트나 Express 등 최신 Framework에 적응하기 위한 필수적인 내용 입니다.

학습목표

> 1. 객체를 이해하고 객체의 개념을 설명할 수 있다.
> 2. Javascript의 class를 이해하고 class 기반으로 객체를 정의할 수 있다.
> 3. 객체의 구성요소를 파악하고 생성자, 멤버변수, 메서드를 정의할 수 있다.
> 4. private 멤버변수를 이해하고 이에 접근하기 위한 getter, setter를 정의할 수 있다.
> 5. 클래스간의 상속을 이해하고 클래스의 기능을 확장하거나 공통 기능을 정의할 수 있다.

#01. 객체 (Object)

- 사전적 의미 : 어떠한 물건이나 대상
- 프로그래밍에서의 의미 : 프로그램에서 표현하고자 하는 기능을 묶기 위한 단위

> 하나의 변수 안에 비슷한 특성을 갖는 변수와 함수가 내장된 형태.

객체를 구성하는 단위

객체를 이루는 것은 데이터와 기능이다.

객체 안에 내장된 변수를 **멤버변수** 혹은 **프로퍼티(속성)** 라고 한다.

객체 안에 내장된 함수를 **메서드**라고 한다.

#02. 클래스 (Class)

객체의 설계도 역할을 하는 프로그램 소스

공장에서 하나의 설계도를 사용하여 여러 개의 제품을 생산할 수 있는 것처럼 하나의 클래스를 통해 동일한 구조를 갖는 객체를 여러 개 생성할 수 있다.

1) 클래스의 가장 기본적인 코드 형식

클래스 이름은 명사들의 조합으로 이루어지며 첫 글자는 대문자로 지정하는 것이 관례이다.

```
class 클래스이름 {
    // 멤버변수 선언
    // 생성자 --> 멤버변수 초기화
    // getter, setter
    // 메서드
}
```

2) 클래스를 통한 객체 생성하기

new 예약어를 사용한다.

```
var|let|const 변수이름 = new 클래스이름();
```

일반적으로 JS에서의 객체 선언은 **const** 키워드를 사용함.

위와 같이 정의하면 변수는 클래스 안에 정의된 모든 기능을 부여받은 특수한 형태의 변수가 되는데 이를 객체라고 하고, 객체는 자신에게 부여된 기능을 점(.)을 통해 접근할 수 있다.

```
객체.멤버변수 = 값;
객체.메서드();
```

3) 클래스의 작성 패턴

1. 변수만 정의
2. 메서드만 정의
3. 변수와 메서드를 함께 정의

객체라는 개념은 배열이 같은 종류의 변수들만 그룹화 하는 한계를 벗어나 서로 다른 종류의 변수를 그룹화 하는데서 출발한다. (이 상태를 C언어의 구조체라고 한다.)

그렇게 그룹화 해 놓은 변수들간의 관계를 구현하기 위해 메서드를 함께 포함하는 형태로 발전된 것이다.

변수만 정의한 클래스

```
class 클래스이름 {
    변수1 = 값;
    변수2 = 값;
    ...
    변수n = 값;
}
```

변수에 값을 초기화 하지 않더라도 멤버변수는 생성된다.

이 경우 객체를 생성하면 멤버변수는 모두 undefined 상태로 존재하기 때문에 객체를 통해 값의 초기화를 별도로 수행해야 한다.

메서드만 정의한 클래스

용도나 목적이 같은 메서드들을 별도의 클래스로 묶어둔다.

```
class 클래스이름 {
    함수이름1(...) { ... }
    함수이름2(...) { ... }
    ...
    함수이름n(...) { ... }
}
```

메서드와 멤버변수를 함께 갖는 클래스

멤버변수의 스코프는 클래스 내의 모든 메서드에서 식별 가능하다. 결국 멤버변수는 모든 메서드가 공유하는 전역 변수의 개념이 된다.

같은 클래스에 속한 멤버변수나 함수끼리는 예약어 **this**를 통해서만 접근 가능하다.

```
class 클래스이름 {
    변수1 = 값;
    변수2 = 값;
    ...
    변수n = 값;

    함수이름1(...) { ... }
    함수이름2(...) { ... }
    ...
    함수이름n(...) { ... }
}
```

하나의 생성자를 통해 동일한 구조를 갖는 객체를 여러개 생성한 예

같은 클래스를 통해 할당된 객체는 동일한 자료 구조를 갖지만 각각 다른 정보를 저장할 수 있다.

생성자 함수

`new` 예약어를 사용하여 인스턴스가 생성될 때 자동으로 실행되는 특수한 함수로서 주로 멤버변수의 값을 초기화 하기 위해 사용한다.

함수 이름이 `constructor()`로 예약되어 있다.

필요에 따라 파라미터를 정의할 수 있으며 파라미터는 주로 멤버변수와 1:1로 대응된다.

클래스 레벨에서 멤버변수를 초기화 할 경우 객체를 생성하면서 멤버변수의 값을 변경할 수 없지만 생성자를 사용하면 객체 생성 단계에서 멤버변수의 값을 다양하게 변경할 수 있다.

```
class 클래스이름 {
    변수1;
    변수2;
    ...
    변수n;

    constructor(파라미터1, 파라미터2, ..., 파라미터n) {
        this.변수1 = 파라미터1;
        this.변수2 = 파라미터2;
        ...
        this.변수n = 파라미터n;
    }
}
```

#03. 은닉성

1) private

객체지향에서는 객체를 통한 멤버변수의 직접 접근이 멤버변수에 값을 대입하는 과정에서 그 값의 적절성을 판단할 수 없고, 무조건적으로 대입하기 때문에 코드 보안에 부적절하다고 보기 때문에 멤버변수나 메서드가 객체를 통해 접근할 수 없도록 클래스 내부에 숨기는 기법이 존재한다.

은닉된 멤버변수와 메서드를 각각 private 프로퍼티(혹은 멤버변수), private 메서드 라고 한다.

멤버변수나 메서드 이름 앞에 `#`을 붙여 적용한다.

클래스를 작성하는 개발자A와 이를 활용하는 개발자B 두 명이 프로젝트를 진행한다고 할 때 개발자A가 개발자B의 실수를 방지하기 위해 클래스 내부에서만 사용할 목적으로 만든 자원을 private으로 설정할 수 있다.

2) getter, setter

멤버변수에 값을 간접적으로 대입하는 특수한 형태의 함수를 `setter`, 멤버변수의 값을 리턴받기 위해 사용하는 특수한 형태의 함수를 `getter`라고 한다.

getter, setter는 일반 메서드와 구분되어야 하며 getter, setter를 정의하기 위한 일반 메서드와는 구별되는 별도의 구문형식이 존재한다.

getter, setter를 사용하면 프로퍼티에 값을 할당하기 전, 값의 적절성을 판단하는 처리과정을 추가할 수 있다.

- 생성자: 객체를 생성할 때 프로퍼티의 값을 초기화 하는 용도
- getter: 객체가 저장하고 있는 프로피터의 값을 조회하는 용도
- setter: 객체가 저장하고 있는 프로퍼티의 값을 수정하는 용도

> 객체지향이 바라보는 관점에서 정리하자면, 모든 멤버변수는 private으로 설정하고 getter, setter를 통해 접근하도록 코드를 작성하는 것이 올바른 코드이다.

1) getter, setter 정의하기

```
class 클래스이름 {
    #멤버변수1;
    #멤버변수2;
    ...
    #멤버변수n;

    set 멤버변수1(value) { this.#멤버변수1 = value; }
    get 멤버변수1() { return this.#멤버변수1; }

    set 멤버변수2(value) { this.#멤버변수2 = value; }
    get 멤버변수2() { return this.#멤버변수2; }

    set 멤버변수n(value) { this.#멤버변수n = value; }
    get 멤버변수n() { return this.#멤버변수n; }
}
```

2) getter, setter 활용하기

함수이지만 변수처럼 사용한다.

```
const 객체 = new 클래스이름();

// setter를 호출한다. 대입되는 값은 setter에 전달되는 파라미터.
객체.함수이름 = 000;

// getter를 호출한다. 멤버변수를 대입하는 것 같지만 실제로는 getter를 호출해서 리턴값을 받는 과정이다.
const 변수 = 객체.함수이름;
```

#04. 클래스 상속

어떤 클래스의 기능을 다른 클래스에 상속시킨 후 추가적인 기능을 명시하여 원래의 기능을 확장하는 방법.

class를 정의할 때 클래스 이름 뒤에 extends 키워드를 명시하고 상속받고자 하는 부모 클래스의 이름을 지정한다.

1) 기능의 확장으로서의 상속

2) 여러 클래스간의 공통 기능을 모아 놓는 의미로서의 상속

여러 개의 클래스가 포함하는 기능 중 일부가 동일한 경우 각 클래스로부터 공통되는 부분을 독립적인 클래스로 추출하고 그 클래스를 상속하여 공유하는 처리 기법.

공통기능을 정의하는 부모 클래스

부모를 상속받는 자식 클래스(들) 정의

자식 클래스에 대한 객체 생성

부모가 생성자 파라미터를 통해 초기화를 수행하고 있다면 그 생성자는 자식 클래스에게도 상속된다.

그러므로 자식 클래스를 통한 객체 생성시에도 부모가 요구하는 생성자 파라미터를 전달해야 한다.

3) 메서드 오버라이드(Override)

클래스 간에 부모-자식 관계가 형성되었을 때 자식 클래스에서 부모 클래스가 갖는 메서드와 동일한 이름의 메서드를 정의하는 기법.

자식이 정의한 메서드에 의해 부모 메서드는 가려지게 된다.

상속 후 자식이 메서드를 추가하는 것이 기능의 확장이라면 메서드 오버라이드는 부모의 기능을 수정하는 개념이다.

4) super 키워드

Override 이전의 원본 기능 호출하기

this 키워드가 현재 클래스나 부모로부터 상속 받은 자원을 가리키는 예약어인 반면, **super** 키워드는 부모의 메서드를 Override 하고 있는 자식 클래스 안에서 부모의 원래 기능을 호출하고자 하는 경우에 사용한다.

부모 클래스의 생성자

super 키워드를 메서드처럼 사용할 경우 부모 클래스의 생성자를 의미한다.

자신의 생성자를 통해 전달받은 파라미터와 추가적으로 가공된 파라미터를 부모의 생성자로 전달하여 객체를 생성하는 방법에 변화를 주고자 할 경우 사용한다.

Class 기반 객체지향 연습문제

문제1.

국어, 영어, 수학 점수를 생성자 파라미터로 입력받아서 합계와 평균을 구하는 클래스 Student를 작성하시오.

이 때 Stuent 클래스는 합계를 리턴하는 메서드인 `sum()`과 평균을 리턴하는 `avg()`를 제공합니다.

작성된 클래스를 활용하여 아래 표에 대한 학생별 합계 점수와 평균점수를 출력하시오.

클래스는 JSON 형식으로 작성되어야 합니다.

이름	국어	영어	수학
철수	92	81	77
영희	72	95	98
민혁	80	86	84

출력결과

```
철수의 총점은 250점 이고 평균은 83.33333333333333점 입니다.
영희의 총점은 265점 이고 평균은 88.33333333333333점 입니다.
민혁의 총점은 250점 이고 평균은 83.33333333333333점 입니다.
```

문제2.

가로(`width`), 세로(`height`)정보를 getter, setter로 관리하는 Rectangle 클래스를 정의하시오.

이 클래스는 생성자의 파라미터가 없으며 둘레의 길이를 구해 리턴하는 getAround() 메서드와 넓이를 구해 리턴하는 gerArea() 메서드를 제공합니다.

클래스는 JSON 형식으로 작성되어야 합니다.

출력결과

가로가 10이고 세로가 5인 경우

```
둘레의 길이는 30이고 넓이는 50입니다.
```

문제3.

다음을 만족하는 Student 클래스를 작성하시오.

1. String형의 학과와 정수형의 학번을 프로퍼티로 선언후 생성자를 통해 주입
2. getter, setter를 정의
3. sayHello() 메서드를 통해 "나는 OOOO학과 OO학번 입니다." 를 출력하는 기능을 구현

클래스 작성 후 아래의 소스를 실행하여 동일한 출력결과를 생성하시오.

```
const stud = new Student("컴퓨터", 202004123);
stud.sayHello();
```

출력결과

```
나는 컴퓨터학과 202004123학번 입니다.
```

문제4.

다음을 만족하는 클래스 Account를 작성하시오.

1. 다음의 2 개의 필드를 선언
   - 문자열 owner; (이름)
   - 숫자형 balance; (금액)
2. 위 모든 필드에 대한 getter와 setter의 구현
3. 위 모든 필드를 사용하는 가능한 모든 생성자의 구현
4. 메소드 deposit()의 헤드는 다음과 같으며 인자인 금액을 저축하는 메소드
   - deposit(amount)
5. 메소드 withdraw()의 헤드는 다음과 같으며 인자인 금액을 인출(리턴)하는 메소드
   - withdraw(long amount)
   - 인출 상한 금액은 잔액까지로 하며, 이 경우 이러한 상황을 출력

클래스 작성 후 아래의 소스를 실행하여 동일한 출력결과를 생성하시오.

```
const acc = new Account("Hello", 15000);
console.log("%s의 잔액은 %d원", acc.owner, acc.balance);

acc.disposit(5000);
console.log("%s의 잔액은 %d원", acc.owner, acc.balance);

acc.withdraw(15000);
console.log("%s의 잔액은 %d원", acc.owner, acc.balance);
```

```
acc.disposit(5000);
console.log("%s의 잔액은 %d원", acc.owner, acc.balance);

acc.withdraw(15000);
console.log("%s의 잔액은 %d원", acc.owner, acc.balance);
```

출력결과

```
Hello의 잔액은 15000원
Hello의 잔액은 20000원
Hello의 잔액은 5000원
Hello의 잔액은 10000원
잔액이 부족합니다.
Hello의 잔액은 10000원
```

문제5

Stack은 배열을 내장하는 클래스로서 FILO(First Input Last Output, 선입후출) 기능을 구현하는 대표적인 자료구조 중 하나이다.

아래의 요구사항을 충족하는 MyList 클래스를 정의하시오.

1. 자료를 저장하기 위한 배열인 data와 배열의 원소 수를 카운트 하기 위한 size라는 멤버변수를 은닉된 형태로 선언한다.
2. 생성자에서는 data를 원소가 0개인 빈 배열로, size는 0으로 초기화 한다.
3. data와 size에 대한 getter는 갖지만 setter는 갖지 않는다.
4. push(item) 메서드는 파라미터로 전달된 값을 배열의 맨 뒤에 추가하고 size의 값을 1 증가시킨다.
5. pop() 메서드는 배열의 마지막 원소를 꺼내어 리턴하고 배열의 크기를 1 축소시킨다.
    - 이를 위해 data는 임시 변수에 깊은 복사 처리된 후 기존의 크기보다 1 작은 사이즈로 새로 초기화 되어야 한다
    - 깊은 복사 처리된 임시 변수의 원소 중에서 마지막 원소를 제외한 상태로 다시 data에 깊은 복사 처리되어야 한다.
    - 이 모든 과정은 slice() 등의 javascript 내장 함수를 사용하지 않고 반복문으로 직접 구현하시오.
6. 완성된 클래스는 아래의 테스트 코드를 사용하여 결과를 확인하시오.

테스트 코드

```
const list = new MyList();

list.push(100);
list.push(200);
list.push(300);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const x = list.pop();
```

```
console.log('추출된 데이터: %d', x);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.push(400);
list.push(500);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const y = list.pop();
console.log('추출된 데이터: %d', y);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.push(600);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const z = list.pop();
console.log('추출된 데이터: %d', z);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);
```

출력결과

```
원소의 수: 3, 데이터 확인: [ 100, 200, 300 ]
추출된 데이터: 300
원소의 수: 2, 데이터 확인: [ 100, 200 ]
원소의 수: 4, 데이터 확인: [ 100, 200, 400, 500 ]
추출된 데이터: 500
원소의 수: 3, 데이터 확인: [ 100, 200, 400 ]
원소의 수: 4, 데이터 확인: [ 100, 200, 400, 600 ]
추출된 데이터: 500
원소의 수: 3, 데이터 확인: [ 100, 200, 400 ]
```

문제6

Queue(큐)는 배열을 내장하는 클래스로서 Stack과 더불어 가장 널리 사용되는 자료구조이다.active

Stack의 특징이 FILO(First Input Last Output, 선입후출)인 반면 Queue의 특징은 FIFO(First Input First Output, 선입선출)이다.

전통적인 자료구조에서는 추가되는 데이터는 무조건 배열의 맨 마지막 원소로 등록되지만 요즘 프로그래밍 언어는 배열의 맨 앞에 데이터를 추가하고 기존의 데이터는 한칸씩 뒤로 밀리는 기능도 제공되고 있다.

문제05에서 완성한 MyList 클래스에 기능을 추가하시오.

1. 문제05에서 구현한 MyList 클래스를 문제06에 동일하게 복사하고 shift() 메서드와 unshift(item) 메서드를 추가한다.
2. shift() 메서드는 data 배열의 가장 첫 번째 원소(index=0)를 꺼내어 리턴하고 배열의 크기를 1축소 시킨다.
    - 이를 위해 data 배열에서 인덱스가 0인 첫 번째 원소를 지역변수(혹은 상수)에 복사한다.
    - 임시 변수를 data의 길이(=size)보다 1작은 크기로 초기화 하고 data 배열에서 인덱스가 1인 두번째 원소부터 나머지 원소들을 임시 변수에 깊은 복사 처리한다. (data[1]이 임시변수[0]에 복사하는

형태)
    - 복사가 완료되면 data의 크기를 기존보다 1작게 다시 초기화 하고 임시 변수의 항목들을 그대로 깊은 복사 처리한다.
    - 이 모든 과정은 slice() 등의 javascript 내장 함수를 사용하지 않고 반복문으로 직접 구현하시오.
3. unshift(item) 메서드는 파라미터로 전달된 값을 data의 맨 첫 번째 원소로 추가하고 기존의 데이터들은 한 칸씩 뒤로 밀어낸다.
    - 이를 위해 임시 변수를 data와 동일한 길이로 생성하고 data의 모든 원소를 깊은 복사 한다.
    - data를 기존의 길이보다 1큰 값으로 초기화 한다.
    - data[0]에 item을 저장한다.
    - data[1]부터는 복사된 임시 변수의 원소를 깊은 복사 처리한다. (임시변수[0]을 data[1]에 복사하는 형태)
    - 이 모든 과정은 slice() 등의 javascript 내장 함수를 사용하지 않고 반복문으로 직접 구현하시오.
4. 완성된 클래스는 아래의 테스트 코드를 사용하여 결과를 확인하시오.

테스트 코드

```
const list = new MyList();

list.push(100);
list.push(200);
list.push(300);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const x = list.shift();
console.log('추출된 데이터: %d', x);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.push(400);
list.push(500);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const y = list.shift();
console.log('추출된 데이터: %d', y);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.push(600);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

const z = list.shift();
console.log('추출된 데이터: %d', z);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.unshift(700);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);

list.unshift(800);
list.unshift(900);
console.log('원소의 수: %d, 데이터 확인: %s', list.size, list.data);
```

출력결과

```
원소의 수: 3, 데이터 확인: [ 100, 200, 300 ]
추출된 데이터: 100
원소의 수: 2, 데이터 확인: [ 200, 300 ]
원소의 수: 4, 데이터 확인: [ 200, 300, 400, 500 ]
추출된 데이터: 200
원소의 수: 3, 데이터 확인: [ 300, 400, 500 ]
원소의 수: 4, 데이터 확인: [ 300, 400, 500, 600 ]
추출된 데이터: 300
원소의 수: 3, 데이터 확인: [ 400, 500, 600 ]
원소의 수: 4, 데이터 확인: [ 700, 400, 500, 600 ]
원소의 수: 6, 데이터 확인: [ 900, 800, 700, 400, 500, 600 ]
```