

# 클래스와 객체

우아한테크코스 5기 FE 가브리엘



# 대부분의 객체 지향 언어에서는...



클래스



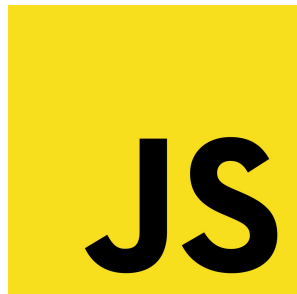
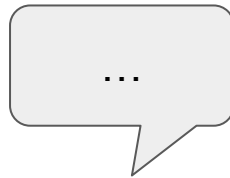
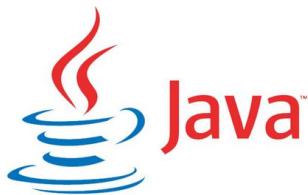
우린 서로  
다른 맛이에요

인스턴스

누구나 들어봤을 법한 유명한 붕어빵 예시 ...

# 대부분의 객체 지향 언어에서는...

클래스가 없다고...?



꼭 클래스가 있어야 객체를 쓸 수 있는 언어일까?

# 우리 자바스크립트는요...

## ECMAScript Editions

Ver	Official Name	Description
ES1	ECMAScript 1 (1997)	First edition
ES2	ECMAScript 2 (1998)	Editorial changes
ES3	ECMAScript 3 (1999)	Added regular expressions Added try/catch Added switch Added do-while
ES4	ECMAScript 4	Never released
ES5	ECMAScript 5 (2009)	Added "strict mode" Added JSON support Added String.trim() Added Array.isArray() Added Array iteration methods Allows trailing commas for object literals

ES6	ECMAScript 2015	Added let and const Added default parameter values Added Array.find() Added Array.findIndex()
	ECMAScript 2016	Added exponential operator (**) Added Array.includes()
	ECMAScript 2017	Added string padding Added Object.entries() Added Object.values() Added async functions Added shared memory Allows trailing commas for function parameters
	ECMAScript 2018	Added rest / spread properties Added asynchronous iteration Added Promise.finally() Additions to RegExp
	ECMAScript 2019	String.trimStart() String.trimEnd() Array.flat() Object.fromEntries Optional catch binding
	ECMAScript 2020	The Nullish Coalescing Operator (??)

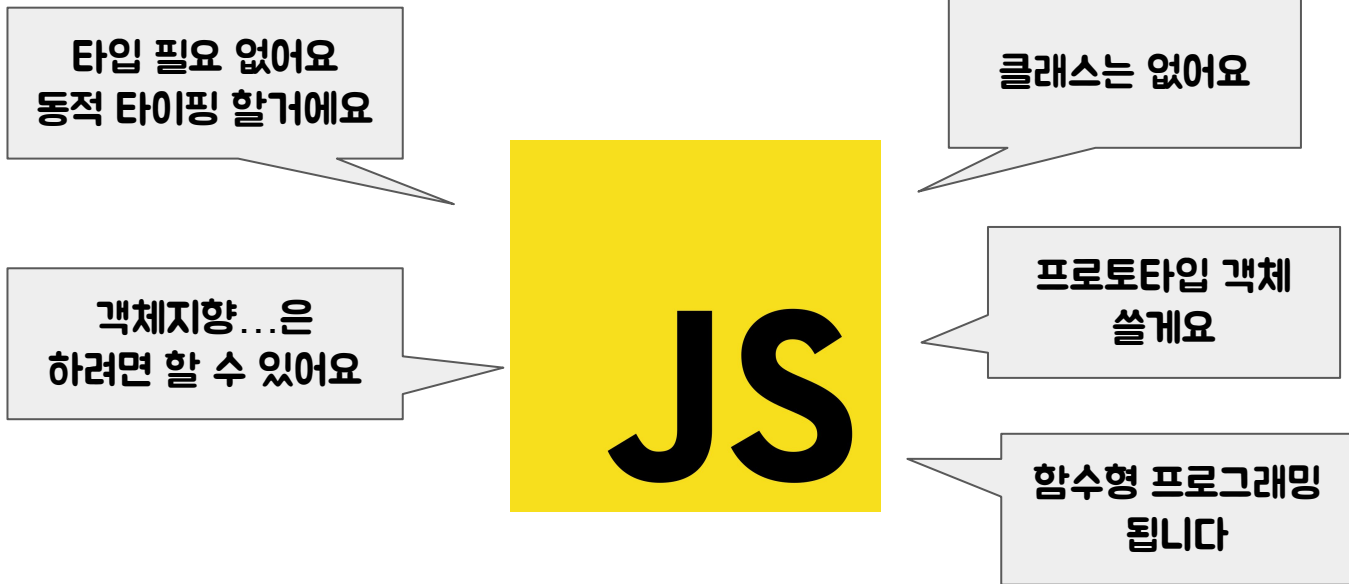
ES6(2015)부터 Classes 문법을 지원하기 시작했다.

# 자바스크립트의 태생을 알아야 한다.



러닝 커브를 낮추기 위해 당시에 유행하고 있던 C++, Java의 문법을 빌려오려고 구상했지만  
**최대한 간단히 만들기 위해서 언어의 복잡도를 최소화 하는 방향으로 구성**

# 자바스크립트는 프로토타입 객체 기반 함수형 동적 타입 스크립트다



굉장히 독특한 컨셉의 언어가 만들어졌다

# 자바스크립트는 많은 오해를 받아왔다



클래스가 없는데  
객체를 어떻게 쓰지?

private가 없는데?

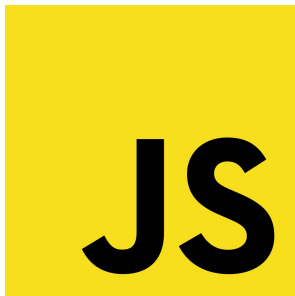


혼란스럽다 ...

그만큼 다른 언어를 쓰던 개발자들의 유입이 어려웠다.

# 결국 자바스크립트도 클래스가 생겼다!

이제 클래스 지원 합니다



근데 여전히 프로토타입 기반임

ES6(2015)부터 Classes 문법을 지원하기 시작했다.



# 클래스 소개에 들어가기 앞서...

객체 문법과 프로토타입에 관한 내용은 간소화했습니다.

# 자바의 클래스 사용법

```
Main.java ×
no usages
1 ▶ public class Main {
    no usages
2 ▶ public static void main(String[] args) {
3     Car car = new Car( name: "JavaCar");
4     System.out.println(car.name + " " + car.position);
5     car.move();
6     System.out.println(car.name + " " + car.position);
7 }
8 }
9
```

```
Main ×
"C:\Program Files\JetBrains\IntelliJ IDEA 2022.3.1\jb
JavaCar 0
JavaCar 1

Process finished with exit code 0
```

```
Car.java ×
2 usages
1 public class Car {
    3 usages
2     String name;
    4 usages
3     int position;
    1 usage
4
5     public Car(String name) {
6         this.name = name;
7         this.position = 0;
8     }
    1 usage
9
10    public void move() {
11        position += 1;
12    }
13 }
```

# 자바스크립트의 클래스 사용법

```
class ClassCar {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}  
  
const classCar = new ClassCar('classCar');  
console.log(classCar); // ClassCar { name: 'classCar', position: 0 }  
classCar.move();  
console.log(classCar); // ClassCar { name: 'classCar', position: 1 }
```

표기법 자체는 자바와 굉장히 유사한 형태다.

## 클래스가 등장하기 전의 생성자 함수를 사용한 객체 생성

```
const FunctionCar = (function () {  
  function FunctionCar(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  FunctionCar.prototype.move = function () {  
    this.position += 1;  
  };  
  
  return FunctionCar;  
})();  
  
const funCar = new FunctionCar('funCar');  
console.log(funCar); // FunctionCar { name: 'funCar', position: 0 }  
funCar.move();  
console.log(funCar); // FunctionCar { name: 'funCar', position: 1 }
```

클래스 없이도 동일한 역할을 하는 객체를 생성할 수 있다.

# 자바스크립트의 클래스와 생성자 함수의 차이점

new 없으면 에러

```
class ClassCar {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}  
  
const classCar = new ClassCar('classCar');  
console.log(classCar); // ClassCar { name: 'classCar', position: 0 }  
classCar.move();  
console.log(classCar); // ClassCar { name: 'classCar', position: 1 }
```

클래스(ES6)

항상  
use strict 모드

extends와 super 지원

new 없이 호출하면  
일반 함수로 취급

```
const FunctionCar = (function () {  
  function FunctionCar(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  FunctionCar.prototype.move = function () {  
    this.position += 1;  
  };  
  return FunctionCar;  
})();  
  
const funCar = new FunctionCar('funCar');  
console.log(funCar); // FunctionCar { name: 'funCar', position: 0 }  
funCar.move();  
console.log(funCar); // FunctionCar { name: 'funCar', position: 1 }
```

생성자 함수(ES5)

결국  
프로토타입으로 변환됨

둘은 사실 많은 관련이 있다.

# 클래스 생성

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메소드  
    this.position += 1;  
  }  
  
  honk() { // 메소드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

클래스 앞에 new를 붙여주면 인스턴스화가 가능하다.

# 클래스 생성

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메소드  
    this.position += 1;  
  }  
  
  honk() { // 메소드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

생성자(constructor) 메서드에서 초기 값을 멤버 변수에 할당하게 된다.

# 클래스 생성

```
// 클래스는 함수입니다.  
console.log(typeof Car); // function  
  
// 명확하게 말해서 클래스는 생성자 메서드와 동일합니다.  
console.log(Car === Car.prototype.constructor); // true  
  
// 클래스 내부에서 정의한 메서드는 Car.prototype에 저장됩니다. (일반적인 메서드는 프로퍼티에 저장)  
console.log(Car.prototype.move); // [Function: move]  
  
// 현재 프로토타입의 메서드를 조회할 수 있습니다.  
console.log(Object.getOwnPropertyNames(Car.prototype)); // [ 'constructor', 'move', 'honk' ]
```

클래스는 결국 함수이고, 메서드는 static 키워드를 붙이지 않는 이상 프로토타입에 연결된다.



# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메서드  
    this.position += 1;  
  }  
  
  honk() { // 메서드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

없어도 된다.

클래스 내부에는 메서드가 0개 이상 존재할 수 있다.

# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메소드  
    this.position += 1;  
  }  
  
  honk() { // 메소드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

클래스 내부에는 **생성자 메서드**, **프로토타입 메서드**, **정적 메서드** 3가지가 올 수 있다.

# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메서드  
    this.position += 1;  
  }  
  
  honk() { // 메서드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

1개 이하만 올 수 있다.

생성자가 2개 이상 오는 경우 에러가 발생한다.

# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메서드  
    this.position += 1;  
  }  
  
  honk() { // 메서드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

이름 변경 불가능!

암묵적으로 this를 반환하므로  
return 금지!

인스턴스를 생성하고 초기화하는데 사용된다.

# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메서드  
    this.position += 1;  
  }  
  
  honk() { // 메서드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

특별한 명시가 없으면  
프로토타입에 저장된다.

move() 메서드는 사실 `FunctionCar.prototype.move = function () {}` 처리된다.

# 메서드

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() { // 메서드  
    this.position += 1;  
  }  
  
  honk() { // 메서드  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');
```

프로토타입 체인에 포함

클래스가 생성한 인스턴스는 생성자 함수와 같이 프로토타입 체인에 포함된다.

# 메서드



```
class ClassCar {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}  
  
const classCar = new ClassCar('classCar');  
console.log(classCar); // ClassCar { name: 'classCar', position: 0 }  
classCar.move();  
console.log(classCar); // ClassCar { name: 'classCar', position: 1 }
```

## 클래스(ES6)

```
const FunctionCar = (function () {  
  function FunctionCar(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  FunctionCar.prototype.move = function () {  
    this.position += 1;  
  };  
  return FunctionCar;  
})();  
  
const funCar = new FunctionCar('funCar');  
console.log(funCar); // FunctionCar { name: 'funCar', position: 0 }  
funCar.move();  
console.log(funCar); // FunctionCar { name: 'funCar', position: 1 }
```

## 생성자 함수(ES5)

둘 다 같은 생성 방식으로 적용되어 **프로토타입 기반의 객체 생성 매커니즘**을 가진다.

# 정적 메서드

```
class Car {  
  static staticMethod() {  
    console.log(this === Car);  
  }  
}  
  
Car.staticMethod(); // true
```

메서드 앞에 `static`을 입력하면 정적 메서드가 된다.

정적 메서드는 프로토타입이 아닌 클래스 함수 자체에 설정되며, 인스턴스를 생성하지 않아도 호출할 수 있게된다.



## 정적 메서드의 활용

```
class Car {  
  constructor(name, date) {  
    this.name = name;  
    this.date = date;  
  }  
  
  static createCar() {  
    // this는 Car입니다.  
    return new this('가브리엘', new Date());  
  }  
}  
  
const car = Car.createCar();  
  
console.log(car); // Car { name: '가브리엘', date: 2023-02-21T18:03:43Z }
```

클래스를 new 키워드(생성자) 없이 정적 처리 하기 위해 사용된다.

# getter/setter

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  get name() {  
    return this._name;  
  }  
  
  set name(newName) {  
    this._name = newName;  
  }  
}  
  
const car = new Car('가브리엘');  
console.log(car); // Car { _name: '가브리엘', position: 0 }  
console.log(car.name); // 가브리엘  
car.name = '엘리브가';  
console.log(car.name); // 엘리브가
```

접근자 프로퍼티로 get과 set 키워드를 지원한다.

# getter/setter

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }  
  
  get name() {  
    return this._name  
  }  
  
  set name(newName) {  
    this._name = newName;  
  }  
}
```

```
const car = new Car('가브리엘');  
console.log(car); // Car { _name: '가브리엘', position: 0 }  
console.log(car.name); // 가브리엘  
car.name = '엘리브가';  
console.log(car.name); // 엘리브가
```

왜 underscore가 나오지?

position과 달리, name 변수는 \_name으로 관리된다. 왜일까?

# getter/setter

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
    this.position = 0;  
  }
```

사실 나도 set 함수 호출하고 있었어

```
  get name() {  
    return this._name;  
  }
```

```
  set name(newName) {  
    this._name = newName;  
  }  
}
```

```
const car = new Car('가브리엘');  
console.log(car); // Car { _name: '가브리엘', position: 0 }  
console.log(car.name); // 가브리엘  
car.name = '엘리브가';  
console.log(car.name); // 엘리브가
```

constructor에서도 setter에 접근을 시도한다.

# getter/setter (오류)

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
  }  
  
  get name() {  
    return this.name;  
  }  
  
  set name(newName) {  
    this.name = newName;  
  }  
}  
  
const car = new Car('가브리엘');
```

```
lass-getter-setter-error.js:11
```

```
    this.name = newName;  
      ^
```

RangeError: Maximum call stack size exceeded

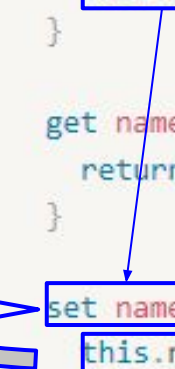
```
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
    at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t
```

Node.js v18.14.0

underscore를 없애면 오류가 난다. 왜일까?

# getter/setter (오류)

```
class Car {  
  constructor(name) { // 생성자  
    this.name = name;  
  }  
  
  get name() {  
    return this.name;  
  }  
  
  set name(newName) {  
    this.name = newName;  
  }  
}  
  
const car = new Car('가브리엘');
```



lass-getter-setter-error.js:11

```
  this.name = newName;  
    ^
```

RangeError: Maximum call stack size exceeded

```
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t  
at set name [as name] (/Users/gabrielyoon7/Code/wooteco_lv1/javascript-for-t
```

Node.js v18.14.0

setter 내부의 this.name도 자기 자신을 호출하려고 시도한다 ...

# private를 활용한 getter/setter

없으면  
SyntaxError: Private field '#name' must  
be declared in an enclosing class  
에러 납니다.

```
class Car {  
  #name = '';  
  
  constructor(name) { // 생성자  
    this.name = name;  
  }  
  
  get name() {  
    return this.#name;  
  }  
  
  set name(newName) {  
    this.#name = newName;  
  }  
}  
  
const car = new Car('가브리엘');  
console.log(car); // Car {}  
console.log(car.name); // 가브리엘
```

은닉화 중이라 안보임

차라리 멤버 변수를 은닉화하도록 #을 붙여주는 것이 안전하다.

# 클래스 상속

자바스크립트의 객체에  
**기본적으로 상속**을 해주기 위함

prototype

상속을 통해 기존 클래스를  
확장하여 **새로운 클래스**로 정의하기 위함

classes

프로토타입 상속과 클래스 상속은 명백하게 다른 개념이다.



# 클래스 상속

수퍼클래스/베이스클래스/부모클래스

서브클래스/파생클래스/자식클래스

```
class Vehicle {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}
```

```
class Car extends Vehicle {  
  honk() {  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}
```

```
const car = new Car('가브리엘');  
console.log(car); // Car { name: '가브리엘', position: 0 }  
  
car.move();  
car.move();  
car.move();  
console.log(car); // Car { name: '가브리엘', position: 3 }  
  
car.honk(); // 가브리엘 : 뽕뽕뽕  
  
console.log(car instanceof Car); // true  
console.log(car instanceof Vehicle); // true
```

# 클래스 상속

Car 클래스에서 정의한 적이 없지만  
다 알고있다.

```
class Vehicle {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}  
  
class Car extends Vehicle {  
  honk() {  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');  
console.log(car); // Car { name: '가브리엘', position: 0 }  
  
car.move();  
car.move();  
car.move();  
console.log(car); // Car { name: '가브리엘', position: 3 }  
  
car.honk(); // 가브리엘 : 뽕뽕뽕  
  
console.log(car instanceof Car); // true  
console.log(car instanceof Vehicle); // true
```

# 클래스 상속

car 인스턴스는

Car의 인스턴스이자

Vehicle의 인스턴스다

```
class Vehicle {
  constructor(name) {
    this.name = name;
    this.position = 0;
  }

  move() {
    this.position += 1;
  }
}

class Car extends Vehicle {
  honk() {
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);
  }
}

const car = new Car('가브리엘');
console.log(car); // Car { name: '가브리엘', position: 0 }

car.move();
car.move();
car.move();
console.log(car); // Car { name: '가브리엘', position: 3 }

car.honk(); // 가브리엘 : 뽕뽕뽕뽕

console.log(car instanceof Car); // true
console.log(car instanceof Vehicle); // true
```

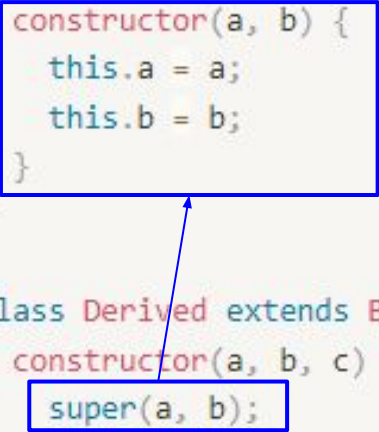
# super (생성자 호출)

생성자 메서드에서

super 키워드를 호출하면

수퍼클래스에 있는 생성자를 호출한다.

```
class Base {  
  constructor(a, b) {  
    this.a = a;  
    this.b = b;  
  }  
}  
  
class Derived extends Base {  
  constructor(a, b, c) {  
    super(a, b);  
    this.c = c;  
  }  
}  
  
const derived = new Derived(1, 2, 3);  
console.log(derived); // Derived { a: 1, b: 2, c: 3 }
```

A blue box highlights the `super(a, b);` line in the `Derived` constructor. A blue arrow points from this box to the `constructor(a, b) {` line in the `Base` class, illustrating that `super` calls the parent class's constructor.

# super (생성자 호출)

자식클래스가 constructor가 없으면  
암묵적으로 constructor에 super가 활성화 되어  
부모클래스의 constructor에 전달된다.

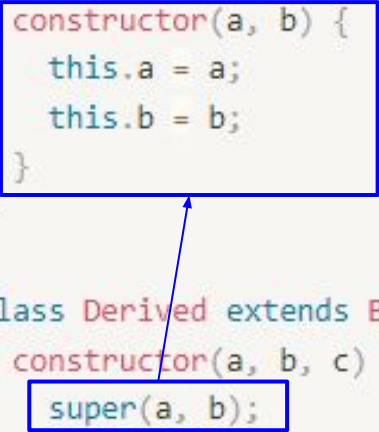
```
class Vehicle {  
  constructor(name) {  
    this.name = name;  
    this.position = 0;  
  }  
  
  move() {  
    this.position += 1;  
  }  
}  
  
class Car extends Vehicle {  
  honk() {  
    console.log(`${this.name} : ${'뽕'.repeat(this.position)}`);  
  }  
}  
  
const car = new Car('가브리엘');  
console.log(car); // Car { name: '가브리엘', position: 0 }  
  
car.move();  
car.move();  
car.move();  
console.log(car); // Car { name: '가브리엘', position: 3 }  
  
car.honk(); // 가브리엘 : 뽕뽕뽕  
  
console.log(car instanceof Car); // true  
console.log(car instanceof Vehicle); // true
```

# super (생성자 호출)

super 키워드는

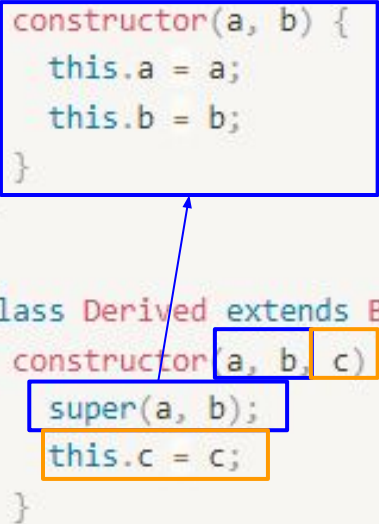
반드시 서브클래스에서만 호출이 가능하다

```
class Base {  
  constructor(a, b) {  
    this.a = a;  
    this.b = b;  
  }  
}  
  
class Derived extends Base {  
  constructor(a, b, c) {  
    super(a, b);  
    this.c = c;  
  }  
}  
  
const derived = new Derived(1, 2, 3);  
console.log(derived); // Derived { a: 1, b: 2, c: 3 }
```



## super (생성자 호출)

```
class Base {  
  constructor(a, b) {  
    this.a = a;  
    this.b = b;  
  }  
}  
  
class Derived extends Base {  
  constructor(a, b, c) {  
    super(a, b);  
    this.c = c;  
  }  
}  
  
const derived = new Derived(1, 2, 3);  
console.log(derived); // Derived { a: 1, b: 2, c: 3 }
```



## super (메서드 참조)

메서드가 바인딩된 객체(수퍼클래스의  
prototype 프로퍼티에 바인딩 된  
프로토타입)여야 참조가 가능

```
class Base {  
  constructor(name) {  
    this.name = name;  
  }  
  
  sayHi() {  
    return `Hi${this.name}`;  
  }  
}  
  
class Derived extends Base {  
  sayHi() {  
    return `${super.sayHi()}`;  
  }  
}  
  
const derived = new Derived('가브리엘');  
console.log(derived.sayHi()); // Hi가브리엘
```



# 상속 클래스의 인스턴스 생성 과정

상속된 파일은 인스턴스를 어떻게 생성할까?

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

  speak() {
    console.log(`${this.name}은(는) 소리를 냅니다.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  speak() {
    super.speak();
    console.log(`${this.name}은(는) 짖습니다.`);
  }

  getBreed() {
    console.log(`${this.name}의 품종은 ${this.breed} 입니다.`);
  }
}

const dog = new Dog('뽀삐', '말티즈');
dog.speak(); // 뽀삐은(는) 소리를 냅니다. 뽀삐은(는) 짖습니다.
dog.getBreed(); // 뽀삐의 품종은 말티즈 입니다.
```

# 상속 클래스의 인스턴스 생성 과정

new 키워드가 Dog의 인스턴스를 생성한다

```
class Animal {
  constructor(name) {
    console.log(1);
    console.log(this); // Dog {}
    this.name = name;
    console.log(2);
    console.log(this); // Dog { name: '뽀삐' }
  }

  speak() {
    console.log(`${this.name}은(는) 소리를 냅니다.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    console.log(3);
    console.log(this); // Dog { name: '뽀삐' }
    this.breed = breed;
    console.log(4);
    console.log(this); // Dog { name: '뽀삐', breed: '말티즈' }
  }

  speak() {
    super.speak();
    console.log(`${this.name}은(는) 짹습니다.`);
  }

  getBreed() {
    console.log(`${this.name}의 품종은 ${this.breed} 입니다.`);
  }
}

const dog = new Dog('뽀삐', '말티즈');
dog.speak(); // 뽀삐은(는) 소리를 냅니다. 뽀삐은(는) 짹습니다.
dog.getBreed(); // 뽀삐의 품종은 말티즈 입니다.
```

# 상속 클래스의 인스턴스 생성 과정

Dog 인스턴스가 전달됐으므로, Animal의 this는 Dog이다

constructor는 암묵적으로 this를 반환하므로

Animal은 Dog를 반환한다.

```
class Animal {
  constructor(name) {
    console.log(1);
    console.log(this); // Dog {}
    this.name = name;
    console.log(2);
    console.log(this); // Dog { name: '뽀삐' }
  }

  speak() {
    console.log(`${this.name}은(는) 소리를 냅니다.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    console.log(3);
    console.log(this); // Dog { name: '뽀삐' }
    this.breed = breed;
    console.log(4);
    console.log(this); // Dog { name: '뽀삐', breed: '말티즈' }
  }

  speak() {
    super.speak();
    console.log(`${this.name}은(는) 짹습니다.`);
  }

  getBreed() {
    console.log(`${this.name}의 품종은 ${this.breed} 입니다.`);
  }
}

const dog = new Dog('뽀삐', '말티즈');
dog.speak(); // 뽀삐은(는) 소리를 냅니다. 뽀삐은(는) 짹습니다.
dog.getBreed(); // 뽀삐의 품종은 말티즈 입니다.
```

# 상속 클래스의 인스턴스 생성 과정

이후에 업데이트를 해도 Dog 인스턴스를 수정해주기 때문에  
Dog 인스턴스가 부모와 자식의 프로퍼티를 모두 가지게  
된다.

```
class Animal {
  constructor(name) {
    console.log(1);
    console.log(this); // Dog {}
    this.name = name;
    console.log(2);
    console.log(this); // Dog { name: '뽀삐' }
  }

  speak() {
    console.log(`${this.name}은(는) 소리를 냅니다.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    console.log(3);
    console.log(this); // Dog { name: '뽀삐' }
    this.breed = breed;
    console.log(4);
    console.log(this); // Dog { name: '뽀삐', breed: '말티즈' }
  }

  speak() {
    super.speak();
    console.log(`${this.name}은(는) 짹습니다.`);
  }

  getBreed() {
    console.log(`${this.name}의 품종은 ${this.breed} 입니다.`);
  }
}

const dog = new Dog('뽀삐', '말티즈');
dog.speak(); // 뽀삐은(는) 소리를 냅니다. 뽀삐은(는) 짹습니다.
dog.getBreed(); // 뽀삐의 품종은 말티즈 입니다.
```

**감사합니다.**