

[JS Deep Dive] 26장 . ES6 함수의 추가 기능



Status

진행 중

ES6 이전과 이후의 비교

함수의 구분



가장 큰 차이 → 함수를 사용 목적에 따라 명확하게 구분 했는지의 유무

▣ ES6 이전



명확한 구분 x → 한 함수를 생성자로서 호출도 가능하고, 일반함수로서 호출할 수 도 있었다.

- 다양한 형태로 함수를 호출
 - 일반함수 호출 / 생성자 함수 호출 / 객체의 메소드로 호출 가능
⇒ 하나의 함수가 여러 역할을 할 수 있었기에 혼란스러웠음
 - 메소드로 만들었는데 실수로 new를 붙여 이상한 객체가 생성되는 문제 발생 가능성이 있었음.

- new를 붙여 이상한 객체가 생성이 되면, 아래 sayHello가 prototype 프로퍼티를 가지고, 프로토 타입 객체도 만들어지게됨.

```
const obj = {
  sayHello: function () {
    console.log("Hello, " + this);
  }
};

obj.sayHello(); // 객체의 메소드로 호출
const newObj = new obj.sayHello(); // 생성자 함수로 호출 -> 엉
//모든 함수가 new를 붙이면 생성자가 될수 있었으므로 혼란스러웠음
```

Callable (호출 가능 함수객체) : → 그냥 실행하면 일반 함수처럼 실행됨

Constructor (인스턴스 생성 가능): → new 를붙이면 생성자 함수로 동작

non-constructor : 인스턴스를 생성할 수 없는 함수 객체

▣ ES6 이후

- ✓ 메소드 단축 문법 도입 → 객체 메소드를 명확하게 구분
- ✓ 화살표 함수 도입 → this 바인딩 문제 해결
- ✓ 클래스(class) 도입 → 생성자 함수와 일반 함수를 구분

ES6 함수의 구분	constructor	prototype	super	arguments
일반 함수(Normal)	○	○	×	○
메서드(Method)	×	×	○	○
화살표 함수(Arrow)	×	×	×	×

메서드

▣ ES6 이전

☑ 일반적으로 객체에 프로퍼티로 할당된 함수 → 명확한 구분기준 x

☑ function 키워드를 사용하여 정의

- 내부슬롯을 가지지 않음
- 구분기준 : 객체의 프로퍼티로 function이 할당된 경우

```
const obj = {  
  name: "Alice",  
  sayHello: function () { //function 키워드로 정의  
    return "Hello, " + this.name;  
  }  
};  
console.log(obj.sayHello()); // Hello, Alice
```

▣ ES6 이후


☑ 단축 메소드 문법 추가



메서드 축약표현으로 정의된 함수만을 의미 ⇒ 정확한 정의 규정



내부슬롯을 가지게 됨 ⇒ ES6의 super 키워드 사용가능

- 내부 슬롯을 가지게 되므로 ES6부터 등장한 내부슬롯을 사용하는 super 키워드를 사용할 수 있게 됨
 -  **super** 키워드 ⇒ 부모 객체의 프로퍼티 or 메소드를 호출할때 사용하는 키워드

```
const parent = {
```

```

    sayHello() { //ES6 단축문법
      return "Hello from Parent";
    }
  };

  const obj = {
    name: "Alice",
    sayHello() {
      return super.sayHello() + `, I'm ${this.name}`;
    }
  };

  // `obj`의 프로토타입을 `parent`로 설정
  Object.setPrototypeOf(obj, parent);

  console.log(obj.sayHello()); // Hello from Parent, I'm Alice

```

화살표 함수

- ✓ ES6에 새로 등장한 함수
- ✓ 기존의 함수 정의방식보다 간략하게 함수 정의
- ✓ 콜백함수 내부에서 `this` 가 전역객체를 가리키는 문제의 대안으로 활용됨.

▣ES6 이전

- 콜백함수내부의 `this` 문제를 해결하기위해 ES6 이전에는 `Array.map` 을 이용해서 두번째 인수로 콜백함수 내부에서 **this** 로 사용할 객체를 전달

▣ES6 이후

- 화살표 함수는 선언된 위치의 **this**를 그대로 상속함
 - ⇒ 콜백함수 내부에서 **this**가 **undefined**가 되는 문제 해결
 - **this** 를 참조하면 스코프 체인을 통해 상위 스코프에서 **this** 를 탐색해야하고, 상위 스코프의 **this**를 가지게 됨
- 함수 자체의 **super** 바인딩을 갖지 않는다.
 - 일반함수에서는 **super**가 호출된 위치에 따라 동적으로 결정되지만, 화살표 함수에서는 상위 스코프에서 **super**를 찾음
 - but 화살표 함수는 자신만의 **arguments** 가 아니라 상위스코프에서 찾음
- 함수 자체의 **arguments** 바인딩을 갖지 않는다
 - 일반 함수에서는 **arguments** 라는 유사배열 객체를 통해 전달된 모든 인자를 접근 가능
- 화살표 함수로 가변인자 함수를 구현해야 할때는 반드시 **rest** 파라미터를 사용해야 한다.
 - 가변인자 함수(인자의 개수가 정해져있지 않은 함수)를 만들때 **arguments** 를 사용할수 없기 때문에, 반드시 **...rest**를 사용해야 한다.

Rest 파라미터

▣ ES6 이전

- rest 파라미터 x
- 가변인자 함수의 경우 **arguments** 객체를 활용했음.
- 그러나 유사배열 객체이므로 배열 메서드를 활용하려면 변환하는 단계를 거쳐야 했음.
- **arguments** 는 화살표 함수에서 사용 x

▣ ES6 이후

- ...rest 는 배열형태로 전달되므로, 배열 메서드를 바로 사용
- 함수 선언 시 몇 개의 인자를 받을지 유연하게 처리 가능

매개변수 기본값

인수가 전달되지 않은 매개변수의 값은 undefined

⇒ 이를 방지하는 방식이 ES6 이전과 이후에 차이가 있음

ES6 이전

- || 연산자 사용 or 삼항 연산자

```
function greet(name) {  
  name = name || "Guest";  
}
```

ES6 이후

- 함수 매개변수에 기본값을 직접 지정

```
function greet(name = "Guest") {  
  console.log(Hello, ${name}!);  
}
```



1 자바스크립트에서 부모 객체(프로토타입)란?

자바스크립트에서 객체는 다른 객체를 상속받을 수 있어.

- 객체를 상속받는다라는 것 = obj 가 다른 객체(parent)를 자신의 프로토타입(부모 객체) 으로 설정할 수 있다는 뜻.
- 프로토타입을 설정하면, obj 에서 없는 프로퍼티나 메소드를 찾을 때 parent 에서 대신 찾게 돼.

2 Object.setPrototypeOf() 가 하는 일

```
js
복사편집
Object.setPrototypeOf(obj, parent);
```

위 코드는 **obj**의 부모 객체(프로토타입)를 **parent**로 설정하는 역할을 해.

즉, **obj**에서 **super**를 사용하면 **parent**의 메소드를 참조하게 됨.

3 **super** 키워드는 어떻게 부모 객체의 메소드를 찾을까?

(1) **super**가 부모 객체의 메소드를 찾는 과정

```
js
복사편집
const parent = {
  sayHello() {
    return "Hello from Parent";
  }
};

const obj = {
  name: "Alice",
  sayHello() {
    return super.sayHello() + `, I'm ${this.name}`;
  }
};

// 부모 객체 설정
Object.setPrototypeOf(obj, parent);

console.log(obj.sayHello());
// 1 obj에 sayHello()가 있으므로 실행됨
// 2 super.sayHello()를 만나면 부모 객체(parent)의 sayHello
//    ()를 호출
// 3 "Hello from Parent, I'm Alice" 반환
```

🔍 요약:

1. `obj.sayHello()` 실행
2. `super.sayHello()` 를 만나서 부모 객체(프로토타입)인 `parent` 의 `sayHello()` 를 호출
3. `"Hello from Parent"` 가 반환되고, `I'm Alice` 를 추가해서 최종 문자열이 출력됨

(2) `obj` 의 프로토타입이 실제로 `parent` 인지 확인하기

자바스크립트에서는 `__proto__` (비표준) 또는 `Object.getPrototypeOf()` 를 사용하면 객체의 부모 객체(프로토타입)를 확인할 수 있어.

```
js
복사편집
console.log(Object.getPrototypeOf(obj) === parent); // true
console.log(obj.__proto__ === parent); // true (비표준이지만 확인 가능)
```

✓ 즉, `obj` 의 프로토타입이 `parent` 로 설정되었기 때문에, `super` 를 사용할 수 있는 것!

🔥 핵심 정리

1. 객체는 프로토타입을 통해 다른 객체를 부모 객체(프로토타입)로 설정할 수 있음
2. `Object.setPrototypeOf(obj, parent)` 를 하면 `parent` 가 `obj` 의 부모 객체가 됨
3. `super.method()` 를 호출하면, `obj` 에 없는 메소드는 `parent` 에서 찾아 실행됨
4. 프로토타입 연결을 확인하려면 `Object.getPrototypeOf(obj) === parent` 를 체크하면 됨