

25. ES6 함수의 추가 기능

TABLE OF CONTENTS

1. 함수의 구분

2. 메소드

3. 화살표 함수

3.1. 화살표 함수 정의

3.2. 화살표 함수와 일반 함수의 차이

3.3. this

3.4. super 빼도 됨

3.5. arguments

4. Rest 파라미터

4.1. 기본 문법

4.2. Rest 파라미터와 arguments 객체

5. 매개변수 기본값

1. 함수의 구분

ES6 이전까지 자바스크립트의 함수는 다른 프로그래밍 언어와는 다르게 별다른 구분없이 다양한 목적으로 사용되었다. 함수는 일반적인 함수로서 호출할 수도 있고, new 연산자와 함께 호출하여 인스턴스를 생성할 수 있는 생성자 함수로서 호출할 수도 있으며, 객체에 바인딩되어 메소드로서 호출할 수도 있다. 이는 언뜻 보면 편리한 것 같지만 실수를 유발시킬 수 있으며 성능면에서도 손해이다.

아래 예제를 살펴보자. ES6 이전의 함수는 동일한 함수라도 다양한 형태로 호출할 수 있다.

JAVASCRIPT

```
var foo = function () {  
  return 1;  
};
```

```
// 일반적인 함수로서 호출  
foo(); // → 1
```

```
// 생성자 함수로서 호출  
new foo(); // → foo {}
```

```
// 메소드로서 호출  
var obj = { a: foo };  
obj.a(); // → 1
```

이처럼 ES6 이전까지 함수는 사용 목적에 따라 명확히 구분되지 않는다. ES6 이전의 함수는 모두 일반 함수로서 호출할 수 있는 것은 물론 생성자 함수로서 호출할 수 있다. 다시 말해, ES6 이전의 모든 함수는 callable이며 constructor이다.

callable과 constructor/non-constructor

“17.2.4. 내부 메소드 `[[Call]]`과 `[[Construct]]`”에서 살펴보았듯이 호출할 수 있는 함수 객체를 callable 이라 하며, 인스턴스를 생성할 수 있는 함수 객체를 constructor, 인스턴스를 생성할 수 없는 함수 객체를 non-constructor라고 부른다.

JAVASCRIPT

```
// foo는 일반 함수이다.  
var foo = function () {};
```

```
// ES6 이전의 모든 함수는 일반 함수로서 호출할 수 있는 것은 물론 생성자 함수로서 호출할 수 있다.  
foo(); // → undefined  
new foo(); // → foo {}
```

주의할 것은 일반적으로 메소드라고 부르는 객체에 바인딩된 함수도 callable이며 constructor이라는 것이다. 따라서 객체에 바인딩된 함수도 일반 함수로서 호출할 수 있는 것은 물론 생성자 함수로서 호출할 수도 있다.

JAVASCRIPT

```
// 프로퍼티 f에 할당된 것은 일반 함수이다.
var obj = {
  x: 10,
  f: function () { return this.x; }
};

// 프로퍼티 f에 할당된 함수를 메소드로서 호출
console.log(obj.f()); // 10

// 프로퍼티 f에 할당된 함수를 일반 함수로서 호출
var bar = obj.f;
console.log(bar()); // undefined

// 프로퍼티 f에 할당된 함수를 생성자 함수로서 호출
console.log(new obj.f()); // f {}
```

위 예제와 같이 객체에 바인딩된 함수를 생성자 함수로 호출하는 경우가 흔치는 않겠지만 문법상 가능하다는 것은 성능면에서도 문제가 있다. 객체에 바인딩된 함수가 constructor이라는 것은 prototype 프로퍼티에 바인딩된 프로토타입 객체를 생성한다는 것을 의미하기 때문이다.

함수에 전달되어 보조 역할만을 수행하는 콜백 함수도 마찬가지이다. 콜백 함수도 constructor이기 때문에 불필요한 프로토타입 객체를 생성한다.

JAVASCRIPT

```
// 콜백 함수를 사용하는 고차 함수 map. 콜백 함수도 프로토타입을 생성한다.
[1, 2, 3].map(function (item) {
  return item * 2;
}); // → [ 2, 4, 6 ]
```

이처럼 ES6 이전의 모든 함수는 사용 목적에 따라 명확한 구분이 없으므로 호출 방식에 특별한 제약이 없고 생성자 함수로 호출되지 않아도 프로토타입 객체를 생성한다. 이는 혼란스러우며 실수를 유발시킬 가능성이 있고 성능에도 좋지 않다.

이러한 문제를 해결하기 위해 ES6에서는 사용 목적에 따라 함수를 3가지 종류로 명확히 구분하였다.

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

일반 함수는 함수 선언문이 함수 표현식으로 정의한 함수를 말하며 ES6 이전의 함수와 차이가 없다. 하지만 ES6의 메소드와 화살표 함수는 ES6 이전의 함수와 명확한 차이가 있다.

일반 함수는 constructor이지만 ES6의 메소드와 화살표 함수는 non-constructor이다. 이에 대해 알아보도록 하자.

2. 메소드

ES6 이전 사양에는 메소드에 대한 명확한 정의가 없었다. 일반적으로 메소드는 객체에 바인딩된 함수를 일컫는 의미로 사용되었다. ES6 사양에서는 메소드에 대한 정의가 명확하게 규정되었다. ES6 사양에서 메소드는 메소드 축약 표현으로 정의된 함수만을 의미한다.

JAVASCRIPT

```
const obj = {
  x: 1,
  // foo는 메소드이다.
  foo() { return this.x; },
  // bar에 바인딩된 함수는 메소드가 아닌 일반 함수이다.
  bar: function() { return this.x; }
};

console.log(obj.foo()); // 1
console.log(obj.bar()); // 1
```

ES6 사양에서 정의한 메소드(이하 ES6 메소드)는 인스턴스를 생성할 없는 non-constructor이다. 따라서 ES6 메소드는 생성자 함수로서 호출할 수 없다.

JAVASCRIPT

```
new obj.foo(); // TypeError: obj.foo is not a constructor
new obj.bar(); // → bar {}
```

ES6 메소드는 인스턴스를 생성할 수 없으므로 prototype 프로퍼티가 없고 프로토타입도 생성하지 않는다.

JAVASCRIPT

```
// obj.foo는 ES6 메소드이므로 prototype 프로퍼티가 없다.
obj.foo.hasOwnProperty('prototype'); // → false

// obj.bar는 일반 함수이므로 prototype 프로퍼티가 있다.
obj.bar.hasOwnProperty('prototype'); // → true
```

참고로 표준 빌드인 객체의 메소드는 모두 non-constructor이다.

JAVASCRIPT

```
console.log(String.prototype.toUpperCase.prototype); // undefined
console.log(Number.prototype.toFixed.prototype); // undefined
console.log(Array.prototype.map.prototype); // undefined
```

ES6 메소드는 메소드가 바인딩된 객체를 가리키는 내부 슬롯 `[[HomeObject]]`를 갖는다. `super` 참조는 내부 슬롯 `[[HomeObject]]`를 사용하여 슈퍼 클래스의 메소드를 참조하므로 내부 슬롯 `[[HomeObject]]`를 갖는 ES6 메소드 만이 `super` 키워드를 사용할 수 있다.

JAVASCRIPT

```
const base = {
  name: 'Lee',
  sayHi() {
    return `Hi! ${this.name}`;
  }
};

const derived = {
  __proto__: base,
```

```
// ES6 메소드이다. [[HomeObject]]를 갖는다.
sayHi() {
  return `${super.sayHi()}. how are you doing?`;
}

console.log(derived.sayHi()); // Hi! Lee. how are you doing?
```

ES6 메소드가 아니면 super 키워드를 사용할 수 없다. 내부 슬롯 [[HomeObject]]를 갖지 않기 때문이다

JAVASCRIPT

```
const derived = {
  __proto__: base,
  // sayHi 프로퍼티의 값은 일반 함수이다. [[HomeObject]]를 갖지 않는다.
  sayHi: function () {
    // SyntaxError: 'super' keyword unexpected here
    return `${super.sayHi()}. how are you doing?`;
  }
};
```

이처럼 ES6에서는 메소드 본연의 기능(super)은 추가하고 의미적으로 맞지 않는 기능(constructor)은 제거하였다. 따라서 메소드를 정의할 때, 프로퍼티 값으로 익명 함수 표현식을 할당하는 ES6 이전의 방식은 더이상 사용하지 않는 것이 좋다.

3. 화살표 함수

화살표 함수(Arrow function)는 function 키워드 대신 화살표(=>, fat arrow)를 사용하여 보다 기존의 함수 정의 방식보다 간략하게 함수를 정의할 수 있다. 화살표 함수는 표현만 간략한 것이 아니라 내부 동작도 기존의 함수보다 간략하다. 특히 화살표 함수는 콜백 함수 내부에서 this가 전역 객체를 가리키는 문제를 해결하기 위한 대안으로 유용하다.

3.1. 화살표 함수 정의

화살표 함수 정의 문법은 아래와 같다.

1. 매개 변수 선언

매개 변수가 여러 개인 경우, 소괄호 () 안에 매개 변수를 선언한다.

JAVASCRIPT

```
(x, y) => { ... }
```

매개 변수가 한 개인 경우, 소괄호 ()를 생략할 수 있다.

JAVASCRIPT

```
x => { ... }
```

매개 변수가 없는 경우, 소괄호 ()를 생략할 수 없다.

JAVASCRIPT

```
() => { ... }
```

[P] 하기 함수 표현식->화살표 함수

2. 함수 몸체 정의

함수 몸체가 한 줄의 문으로 구성된다면 함수 몸체를 감싸는 중괄호 {}를 생략할 수 있다. 이때 문은 암묵적으로 반환된다.

JAVASCRIPT

```
x => x * x;
```

```
// 위 표현과 동일하다.
```

```
x => { return x * x; }
```

```
// 매개 변수가 없는 화살표 함수
```

```
const now = () => Date.now();
```

```
now(); // → 1567061352352
```

```
// 함수 표현식
var now = function() {
  return Date.now();
};
now(); // → 1567061352352

// 매개 변수가 한 개인 화살표 함수 -> 매개변수가 1개->소괄호 생략 가능
// & 함수 몸체가 한줄의 문으로 구성->중괄호 생략 가능
const identity = value => value;

// 함수 표현식
var identity = function(value) {
  return value;
};

// 매개 변수가 여러 개인 화살표 함수
const sum = (a, b) => a + b;

// 함수 표현식
var sum = function(a, b) {
  return a + b;
};
```

함수 몸체가 여러 줄의 문으로 구성된다면 함수 몸체를 감싸는 중괄호 {}를 생략할 수 없다. 이때 반환값이 있다면 명시적으로 반환해야 한다

JAVASCRIPT

```
// 화살표 함수
const sum = (a, b) => {
  const result = a + b;
  return result;
};

// 함수 표현식
var sum = function (a, b) {
  const result = a + b;
  return result;
};
```


객체 리터럴을 반환하는 경우, 객체 리터럴을 소괄호 ()로 감싸 주어야 한다.

JAVASCRIPT

```
() => { return { a: 1 }; }
// 위 표현과 동일하다.
() => ({ a: 1 })

// 화살표 함수
const create = (id, content) => ({ id, content });

// 함수 표현식
var create = function (id, content) {
  return { id, content };
};

create(1, 'JavaScript'); // -> {id: 1, content: "JavaScript"}
```

화살표 함수도 즉시 실행 함수(IIFE)로 사용할 수 있다.

JAVASCRIPT

```
const person = (name => ({
  sayHi() { return `Hi? My name is ${name}`; }
}))('Lee');

console.log(person.sayHi()); // Hi? My name is Lee
```

화살표 함수도 일급 객체이므로 `Array.prototype.map`, `Array.prototype.filter`, `Array.prototype.reduce`와 같은 고차 함수(Higher-Order Function, HOF)에 인수로 전달할 수 있다. 이 경우 일반적인 함수 표현식보다 표현이 간결하다.

JAVASCRIPT

```
// ES5
[1, 2, 3].map(function (v) {
  return v * 2; -> 인수(콜백함수)
});
```

*`Array.prototype.map`
`map` 메소드는 배열을 순회하며 배열의 각 요소에 대하여 인수로 전달된 콜백 함수를 실행한다. 그리고 콜백 함수의 반환값들로 구성된 새로운 배열을 반환한다. 이때 원본 배열은 변경되지 않는다.

```
// ES6
[1, 2, 3].map(v => v * 2); // → [ 2, 4, 6 ]
```

이처럼 화살표 함수는 콜백 함수로서 정의할 때 유용하다. 표현만 간략한 것만이 아니다. 화살표 함수는 일반 함수의 기능을 간략화 했으며 this를 사용하는 것도 편리하게 설계되었다. 일반 함수와의 차이에 대해 살펴보자.

3.2. 화살표 함수와 일반 함수의 차이

화살표 함수와 일반 함수의 차이는 아래와 같다.

1. 화살표 함수는 인스턴스를 생성할 수 없는 non-constructor이다.

따라서 화살표 함수는 생성자 함수로서 호출할 수 없다.

JAVASCRIPT

```
const Foo = () => {};
new Foo(); // TypeError: Foo is not a constructor
```

화살표 함수는 인스턴스를 생성할 수 없으므로 prototype 프로퍼티가 없고 프로토타입도 생성하지 않는다.

JAVASCRIPT

```
const Foo = () => {};
// 화살표 함수 Foo는 prototype 프로퍼티가 없다.
Foo.hasOwnProperty('prototype'); // → false
```

2. 중복된 매개 변수 이름을 선언할 수 없다.

일반 함수는 중복된 매개 변수 이름을 선언해도 에러가 발생하지 않는다.

JAVASCRIPT

```
function normal(a, a) { return a + a; }
console.log(normal(1, 2)); // 4
```

3이 아닌 4가 출력되는 이유는, 매개변수 이름이 중복될 경우 뒤의 매개변수만을 연산한 것인가?

화살표 함수는 중복된 매개 변수 이름을 선언할 수 없다.

JAVASCRIPT

```
const arrow = (a, a) => a + a;
// SyntaxError: Duplicate parameter name not allowed in this context
```

3. 화살표 함수는 함수 자체의 this, arguments, super, new.target 바인딩을 갖지 않는다.

따라서 화살표 함수 내부에서 this, arguments, super, new.target를 참조하면 스코프 체인을 통해 상위 컨텍스트의 this, arguments, super, new.target를 참조한다.

화살표 함수가 화살표 함수의 중첩 함수인 경우, 부모 화살표 함수에도 this, arguments, super, new.target 바인딩이 없으므로 부모 화살표 함수의 상위 컨텍스트의 this, arguments, super, new.target를 참조한다. 즉, 화살표 함수가 중첩 함수인 경우, 상위 스코프에 존재하는 가장 가까운 함수 중에서 화살표 함수가 아닌 부모 함수의 this, arguments, super, new.target를 참조한다..

3.3. this

화살표 함수가 일반 함수와 구별되는 가장 큰 특징은 바로 this이다. 그리고 화살표 함수는 다른 함수의 인수로 전달되어 콜백 함수로 사용되는 경우가 많다.

화살표 함수의 this는 일반 함수의 this와 다르게 동작한다. 이는 “콜백 함수 내부의 this 문제”, 즉 콜백 함수 내부의 this가 외부 함수의 this와 다르기 때문에 발생하는 문제를 해결하기 위해 의도적으로 설계된 것이다. 콜백 함수의 this에 대해 다시 한번 살펴보자.

“22. this”에서 살펴보았듯이 this 바인딩은 함수의 호출 방식, 즉 함수가 어떻게 호출되었는지에 따라 동적으로 결정된다. 다시 말해, 함수를 정의할 때 this에 바인딩할 객체가 정적으로 결정되는 것이 아니고, 함수를 호출할 때 함수가 어떻게 호출되었는지에 따라 this에 바인딩할 객체가 동적으로 결정된다.

이때 주의할 것은 일반 함수로 호출되는 콜백 함수의 경우이다. 어떤 함수의 인수로 전달되어 함수 내부에서 호출되는 콜백 함수도 중첩 함수라고 할 수 있다. 주어진 배열의 각 요소에 접두어를 추가하는 아래 예제를 살펴보자.

JAVASCRIPT

<주어진 배열의 각 요소에 접두어 추가>

```

class Prefixer {
  constructor(prefix) {
    this.prefix = prefix;
  }

  prefixArray(arr) {
    // ① 여기서의 this : 메소드를 호출한 객체 Prefixer
    // 인수로 전달된 배열 arr을 순회하며 배열의 모든 요소에 prefix를 추가한다.
    return arr.map(function (item) { -> Array.prototype.map의 인수로 전달된 콜백함수
      return this.prefix + ' ' + item; // ② 여기(콜백 함수 내부)에서의 this : 전역 객체
    // -> TypeError: Cannot read property 'prefix' of undefined
    });
  }
}

const prefixer = new Prefixer('Hi');
console.log(prefixer.prefixArray(['Lee', 'Kim']));

```

위 예제를 실행했을 때 기대하는 결과는 ['Hi Lee', 'Hi Kim'] 이다. 하지만 TypeError가 발생한다. 그 이유에 대해 살펴보자.

프로토타입 메소드 내부인 ①에서 this는 메소드를 호출한 객체(위 예제의 경우 prefixer 객체)를 가리킨다. 그런데 Array.prototype.map의 인수로 전달한 콜백 함수의 내부인 ②에서 this는 전역 객체를 가리킨다. 이는 콜백 함수가 일반 함수로 호출되기 때문이다. 생성자 함수 또는 메소드로서 호출되지 않고 일반 함수로서 호출되는 모든 함수 내부의 this는 전역 객체를 가리킨다. 하지만 클래스 내부 코드는 모두 strict mode가 적용되고 strict mode에서 함수를 일반 함수로서 호출하면 this에 undefined가 바인딩된다. (“20.6.1. 일반 함수의 this” 참고)

이때 발생하는 문제가 바로 “콜백 함수 내부의 this 문제”이다. 즉, 콜백 함수의 this(②)와 외부 함수의 this(①)가 서로 다른 객체를 가리키고 있기 때문에 TypeError가 발생한 것이다

ES6 이전에는 이와 같은 “콜백 함수 내부의 this 문제”를 해결하기 위해 아래와 같이 3가지 방법을 사용했다.

1. 메소드를 호출한 prefixer 객체를 가리키는 this를 일단 회피시킨 다음 콜백 함수 내부에서 사용한다.

JAVASCRIPT

```
...
prefixArray(arr) {
  const that = this;
  return arr.map(function (item) {
    return that.prefix + ' ' + item;
  });
}
...
```

2. Array.prototype.map의 2번째 매개 변수에 메소드를 호출한 prefixer 객체를 가리키는 this를 전달한다.

ES5에서 도입된 Array.prototype.map은 “콜백 함수 내부의 this 문제”를 해결하기 위해 두번째 매개 변수에 this로 사용할 객체를 전달할 수 있다.

JAVASCRIPT

```
...
prefixArray(arr) {
  return arr.map(function (item) {
    return this.prefix + ' ' + item;
  }, this);
}
...
```

3. Function.prototype.bind 메소드를 사용하여 메소드를 호출한 prefixer 객체를 가리키는 this를 바인딩한다.

JAVASCRIPT

```
...
prefixArray(arr) {
  return arr.map(function (item) {
    return this.prefix + ' ' + item;
  }).bind(this);
}
...
```

ES6에서는 화살표 함수를 사용하여 “콜백 함수 내부의 this 문제”를 해결할 수 있다.

JAVASCRIPT

```
class Prefixer {
  constructor(prefix) {
    this.prefix = prefix;
  }

  prefixArray(arr) {
    return arr.map(item => `${this.prefix} ${item}`);
  }
}

const prefixer = new Prefixer('Hi');
prefixer.prefixArray(['Lee', 'Kim']); // → ['Hi Lee', 'Hi Kim']
```

```
prefixArray(arr) {
  // ①
  // 인수로 전달된 배열 arr을 순회하며 배열의 모든 요소에 prefix를 추가한다.
  return arr.map(function (item) {
    return this.prefix + ' ' + item; // ②
    // → TypeError: Cannot read property 'prefix' of undefined
  });
}

const prefixer = new Prefixer('Hi');
console.log(prefixer.prefixArray(['Lee', 'Kim']));
```

화살표 함수는 함수 자체의 this 바인딩이 없다. 화살표 함수 내부에서 this를 참조하면 상위 컨텍스트의 this를 그대로 참조한다. 이를 **Lexical this**라 한다. 이는 마치 **렉시컬 스코프**와 같이 화살표 함수의 this가 함수가 정의된 위치에 의해 결정된다는 것을 의미한다.

화살표 함수를 제외한 모든 컨텍스트에는 this 바인딩이 반드시 존재한다. 따라서 일반적인 식별자와는 다르게 this는 스코프 체인을 통해 탐색하지 않는다. 아니 탐색할 필요가 없다. 하지만 화살표 함수 내부에는 this가 없다. 따라서 화살표 함수 내부에서 this를 참조하면 스코프 체인을 통해 this의 값을 탐색한다. 화살표 함수를 `Function.prototype.bind`를 사용하여 표현하면 아래와 같다.

JAVASCRIPT

```
// 화살표 함수는 상위 컨텍스트의 this를 참조한다.
() => this.x;
```

```
// 익명 함수에 this를 주입한다. 위 화살표 함수와 동일하게 동작한다.
(function () { return this.x; }).bind(this);
```

만약 화살표 함수가 화살표 함수의 중첩 함수인 경우, 부모 화살표 함수가 참조하는 상위 컨텍스트의 this를 참조한다. 즉, 화살표 함수가 중첩 함수인 경우, 상위 스코프에 존재하는 가장 가까운 함수 중에서 화살표 함수가 아닌 부모 함수의 this를 참조한다. 만약 화살표 함수가 전역 함수라면 화살표 함수의 this는 전역 객체를 가리킨다.

JAVASCRIPT

```
// 화살표 함수는 함수 자체의 this 바인딩이 없다.
// 전역 함수 foo의 상위 컨텍스트는 전역이다.
// 화살표 함수 foo의 this는 전역 객체를 가리킨다.
const foo = () => console.log(this);
foo(); // window

// 중첩 함수 foo의 상위 컨텍스트는 즉시 실행 함수이다.
// 화살표 함수 foo의 this는 즉시 실행 함수의 this를 가리킨다.
(function () {
  const foo = () => console.log(this);
  foo();
}).call({ a: 1 }); // { a: 1 }

// 함수 foo는 화살표 함수를 반환한다.
// 반환된 화살표 함수의 this는 즉시 실행 함수의 this를 가리킨다.
(function () {
  const foo = () => () => console.log(this);
  foo()();
}).call({ a: 1 }); // { a: 1 }

// increase 프로퍼티에 할당한 화살표 함수의 상위 컨텍스트는 전역이다.
// increase 프로퍼티에 할당한 화살표 함수의 this는 전역 객체를 가리킨다.
const counter = {
  num: 1,
  increase: () => ++this.num
};

console.log(counter.increase()); // NaN
```

화살표 함수 내부의 this는 Function.prototype.call, Function.prototype.apply, Function.prototype.bind 메소드를 사용하여 변경할 수 없다.

JAVASCRIPT

```
window.x = 1;

const normal = function () { return this.x; };
const arrow = () => this.x;
```

```
console.log(normal.call({ x: 10 })); // 10
console.log(arrow.call({ x: 10 })); // 1
```

화살표 함수가 `Function.prototype.call`, `Function.prototype.apply`, `Function.prototype.bind` 메소드를 사용할 수 없다는 의미는 아니다. 단지 화살표 함수의 `this`는 일단 결정된 이후 변경할 수 없고 언제나 유지된다.

JAVASCRIPT

```
const add = (a, b) => a + b;

// this 위치에 null을 넣었다는 것은 무슨 의미?
console.log(add.call(null, 1, 2)); // 3
console.log(add.apply(null, [1, 2])); // 3
console.log(add.bind(null, 1, 2)()); // 3
```

메소드를 화살표 함수로 정의하는 것은 피해야 한다. 화살표 함수로 메소드를 정의하여 보자.

JAVASCRIPT

```
// Bad
const person = {
  name: 'Lee',
  sayHi: () => console.log(`Hi ${this.name}`)
};

// 전역 객체 window에는 빌트인 프로퍼티 name이 존재한다.
// sayHi에 할당된 화살표 함수 내부의 this는 전역 객체를 가리키므로
// 이 예제를 브라우저에서 실행하면 빈문자열을 갖는 window.name이 출력된다.
person.sayHi(); // Hi
```

위 예제의 경우, `sayHi` 프로퍼티에 할당한 화살표 함수 내부의 `this`는 메소드를 호출한 객체를 가리키지 않고 상위 컨텍스트인 전역 객체를 가리킨다. 따라서 화살표 함수로 메소드를 정의하는 것은 바람직하지 않다. 따라서 ES6 메소드 정의를 사용하는 것이 좋다.

JAVASCRIPT

```
// Good
const person = {
```



```
name: 'Lee',
sayHi() {
  console.log(`Hi ${this.name}`);
}
};
```

```
person.sayHi(); // Hi Lee
```

프로토타입 객체에 화살표 함수를 할당하는 경우도 동일한 문제가 발생한다.

JAVASCRIPT

```
// Bad
function Person(name) {
  this.name = name;
}

Person.prototype.sayHi = () => console.log(`Hi ${this.name}`);

const person = new Person('Lee');
// 이 예제를 브라우저에서 실행하면 빈문자열을 갖는 window.name이 출력된다.
person.sayHi(); // Hi
```

프로토타입 객체에는 ES6 메소드 정의를 사용할 수 없으므로 일반 함수를 할당한다.

JAVASCRIPT

```
// Good
function Person(name) {
  this.name = name;
}

Person.prototype.sayHi = function () { console.log(`Hi ${this.name}`); };

const person = new Person('Lee');
person.sayHi(); // Hi Lee
```

클래스 필드 정의 제안을 사용하여 클래스 필드에 화살표 함수를 할당할 수도 있다.

JAVASCRIPT

```
// Bad
class Person {
  // 클래스 필드 정의 제안
  name = 'Lee';
  sayHi = () => console.log(`Hi ${this.name}`);
}

const person = new Person();
person.sayHi(); // Hi Lee
```

이때 sayHi 클래스 필드에 할당한 화살표 함수 내부에서 this를 참조하면 상위 컨텍스트의 this를 그대로 가리킨다. 그렇다면 sayHi 클래스 필드에 할당한 화살표 함수의 상위 컨텍스트는 무엇일까? sayHi 클래스 필드는 인스턴스 프로퍼티이므로 아래와 같은 의미이다.

JAVASCRIPT

```
class Person {
  constructor() {
    this.name = 'Lee';
    // 클래스가 생성한 인스턴스(this)의 sayHi 프로퍼티에 화살표 함수를 할당한다.
    // sayHi 프로퍼티는 인스턴스 프로퍼티이다.
    this.sayHi = () => console.log(`Hi ${this.name}`);
  }
}
```

따라서 sayHi 클래스 필드에 할당한 화살표 함수의 상위 컨텍스트는 constructor이며 화살표 함수의 this는 constructor의 this와 같다. constructor의 this는 클래스가 생성한 인스턴스를 가리키므로 sayHi 클래스 필드에 할당한 화살표 함수 내부의 this 또한 클래스가 생성한 인스턴스를 가리킨다.

하지만 클래스 필드에 할당한 화살표 함수는 프로토타입 메소드가 아니라 인스턴스 메소드가 된다. 따라서 ES6 메소드 정의를 사용하는 것이 좋다.

JAVASCRIPT

```
// Good
class Person {
  // 클래스 필드 정의
  name = 'Lee';
```

```

    sayHi() { console.log(`Hi ${this.name}`); }
}
const person = new Person();
person.sayHi(); // Hi Lee

```

3.4. super

화살표 함수는 함수 자체의 super 바인딩이 없다. 따라서 화살표 함수 내부에서 super를 참조하면 상위 컨텍스트의 super를 참조한다.

JAVASCRIPT

```

class Base {
  constructor(name) {
    this.name = name;
  }

  sayHi() {
    return `Hi! ${this.name}`;
  }
}

class Derived extends Base {
  // super 키워드는 ES6 메소드 내에서만 사용 가능하다.
  // 화살표 함수는 함수 자체의 super 바인딩이 없다.
  // 화살표 함수 foo의 상위 컨텍스트는 constructor이다.
  // 화살표 함수 foo의 super는 constructor의 super를 가리킨다.
  // 클래스 필드 정의 제안으로 클래스 필드에 화살표 함수를 할당한다.
  sayHi = () => `${super.sayHi()} how are you doing?`;
}

const derived = new Derived('Lee');
console.log(derived.sayHi()); // Hi! Lee how are you doing?

```

super는 내부 슬롯 [[HomeObject]]를 갖는 ES6 메소드만이 사용할 수 있는 키워드이다. 위 예제의 sayHi 클래스 필드에 할당한 화살표 함수는 ES6 메소드가 아니지만 상위 컨텍스트의 super를 그대로

참조하기 때문에 super 참조가 가능하다.

sayHi 클래스 필드에 할당한 화살표 함수의 상위 컨텍스트는 constructor이며 화살표 함수의 super는 constructor의 super와 같다.

3.5. arguments

화살표 함수는 함수 자체의 arguments 바인딩이 없다. 따라서 화살표 함수 내부에서 arguments를 참조하면 상위 컨텍스트의 arguments를 참조한다.

JAVASCRIPT

```
(function () {  
  // 화살표 함수는 함수 자체의 arguments 바인딩이 없다.  
  // 중첩 함수 foo의 상위 컨텍스트는 즉시 실행 함수이다.  
  // 화살표 함수 foo의 arguments는 실행 함수의 arguments를 가리킨다.  
  const foo = () => console.log(arguments); // [Arguments] { '0': 1, '1': 2  
}  
  foo(3, 4);  
}(1, 2));  
  
// 전역 함수 foo의 상위 컨텍스트는 전역이다.  
// 전역에는 arguments 객체가 없다. arguments 객체는 함수 내부에서만 유효하다.  
const foo = () => console.log(arguments);  
foo(1, 2); // ReferenceError: arguments is not defined
```

“18.2.1. arguments 프로퍼티”에서 살펴본 바와 같이 arguments 객체는 매개변수 개수를 확인할 수 없는 가변 인자 함수를 구현할 때 유용하다. 하지만 화살표 함수에서는 arguments 객체를 사용할 수 없다. 상위 컨텍스트의 arguments 객체를 참조할 수는 있지만 화살표 함수 자신에게 전달된 인수 목록을 확인할 수 없으므로 그다지 도움이 되지 않는다.

따라서 화살표 함수로 가변 인자 함수를 구현해야 할 때는 반드시 Rest 파라미터를 사용해야 한다.

4. Rest 파라미터

4.1. 기본 문법

Rest 파라미터(Rest Parameter, 나머지 매개변수)는 매개변수 이름 앞에 세개의 점 ...을 붙여서 정의한 매개변수를 의미한다. Rest 파라미터는 함수에 전달된 인수들의 목록을 배열로 전달받는다.

JAVASCRIPT

```
function foo( ... rest) {  
  // 매개변수 rest는 인수들의 목록을 배열로 전달받는 Rest 파라미터이다.  
  console.log(rest); // [ 1, 2, 3, 4, 5 ]  
  // 매개변수 rest에는 배열이 할당된다.  
  console.log(Array.isArray(rest)); // true  
}  
  
foo(1, 2, 3, 4, 5);
```

함수에 전달된 인수들은 순차적으로 매개변수와 Rest 파라미터에 할당된다.

JAVASCRIPT

```
function foo(param, ... rest) {  
  console.log(param); // 1  
  console.log(rest); // [ 2, 3, 4, 5 ]  
}  
  
foo(1, 2, 3, 4, 5);  
  
function bar(param1, param2, ... rest) {  
  console.log(param1); // 1  
  console.log(param2); // 2  
  console.log(rest); // [ 3, 4, 5 ]  
}  
  
bar(1, 2, 3, 4, 5);
```

Rest 파라미터는 이름 그대로 먼저 선언된 매개변수에 할당된 인수를 제외한 나머지 인수들이 모두 배열에 담겨 할당된다. 따라서 Rest 파라미터는 반드시 마지막 이어야 한다.

JAVASCRIPT

```
function foo( ...rest, param1, param2) { }
```



```
foo(1, 2, 3, 4, 5);
```



```
// SyntaxError: Rest parameter must be last formal parameter
```

Rest 파라미터는 단 하나만 선언할 수 있다.

JAVASCRIPT

```
function foo( ...rest1, ...rest2) { }
```



```
foo(1, 2, 3, 4, 5);
```



```
// SyntaxError: Rest parameter must be last formal parameter
```

Rest 파라미터는 함수 정의 시 선언한 매개변수 개수를 나타내는 함수 객체의 length 프로퍼티에 영향을 주지 않는다.

JAVASCRIPT

```
function foo( ...rest) {}  
console.log(foo.length); // 0
```



```
function bar(x, ...rest) {}  
console.log(bar.length); // 1
```



```
function baz(x, y, ...rest) {}  
console.log(baz.length); // 2
```

4.2. Rest 파라미터와 arguments 객체


ES5에서는 인자의 개수를 사전에 알 수 없는 가변 인자 함수의 경우, arguments 객체를 통해 인수를 확인한다. arguments 객체는 함수 호출 시 전달된 인수(argument)들의 정보를 담고 있는 순회 가능한(iterable) 유사 배열 객체(array-like object)이며 함수 내부에서 지역 변수처럼 사용할 수 있다.

JAVASCRIPT

```
// 매개변수의 개수를 사전에 알 수 없는 가변 인자 함수
function sum() {
  // 가변 인자 함수는 arguments 객체를 통해 인수를 전달받는다.
  console.log(arguments);
}

sum(1, 2); // {length: 2, '0': 1, '1': 2}
```

가변 인자 함수는 매개변수를 통해 인수를 전달받는 것이 불가능하므로 arguments 객체를 활용하여 인수를 전달받는다. 하지만 arguments 객체는 유사 배열 객체이므로 배열 메소드를 사용하려면 Function.prototype.call 메소드를 통해 this를 변경하여 배열 메소드를 호출해야 하는 번거로움이 있다.



```
function sum() {
  // 유사 배열 객체인 arguments 객체를 배열로 변환한다.
  var array = Array.prototype.slice.call(arguments);

  return array.reduce(function (pre, cur) {
    return pre + cur;
  }, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15
```

* slice 메소드

slice 메소드는 인수로 전달된 범위의 요소들을 복사하여 반환한다. 원본 배열은 변경되지 않는다.

slice 메소드는 2개의 매개변수를 갖는다.

start : 복사를 시작할 인덱스이다. 음수인 경우, 배열의 끝에서의 인덱스를 나타낸다. 예를 들어 slice(-2)는 배열의 마지막 2개의 요소를 반환한다.
end : 복사를 종료할 인덱스이다. 이 인덱스에 해당하는 요소는 복사되지 않는다. 옵션이며 기본값은 length 값이다.

* reduce 메소드

배열을 순회하며 콜백 함수의 이전 반환값과 배열의 각 요소에 대하여 인수로 전달된 콜백 함수를 실행하여 하나의 결과값을 반환한다. 이때 원본 배열은 변경되지 않는다.

reduce 메소드는 첫번째 인수로 콜백 함수, 두번째 인수로 초기값을 전달받는다. reduce 메소드의 콜백 함수에는 4개의 인수, 초기값 또는 콜백 함수의 이전 반환값, 요소값, 인덱스, reduce 메소드를 호출한 배열, 즉 this가 전달된다.

ES6에서는 rest 파라미터를 사용하여 가변 인자의 목록을 배열로 직접 전달받을 수 있다. 이를 통해 유사 배열인 arguments 객체를 배열로 변환하는 번거로움을 피할 수 있다.

JAVASCRIPT

```
function sum( ... args) {
  // Rest 파라미터 args에는 배열 [1, 2, 3, 4, 5]이 할당된다.
  return args.reduce((pre, cur) => pre + cur, 0);
}

console.log(sum(1, 2, 3, 4, 5)); // 15
```

일반 함수와 메소드는 Rest 파라미터와 arguments 객체를 모두 사용할 수 있다. 하지만 화살표 함수는 함수 자체의 arguments 객체를 갖지 않는다. 따라서 화살표 함수로 가변 인자 함수를 구현해야 할 때는 반드시 Rest 파라미터를 사용해야 한다.

JAVASCRIPT

```
var normalFunc = function () {};  
console.log(normalFunc.hasOwnProperty('arguments')); // true  
  
const arrowFunc = () => {};  
console.log(arrowFunc.hasOwnProperty('arguments')); // false
```

5. 매개변수 기본값

함수를 호출할 때 매개변수의 개수만큼 인수를 전달하는 것이 바람직하지만 그렇지 않은 경우에도 에러가 발생하지는 않는다. 이는 자바스크립트 엔진이 함수의 매개변수의 개수와 인수의 개수를 체크하지 않기 때문이다. 인수가 부족한 경우, 매개변수의 값은 undefined이다.

JAVASCRIPT

```
function sum(x, y) {  
  return x + y;  
}  
  
console.log(sum(1)); // NaN
```

따라서 매개변수에 적절한 인수가 전달되었는지 함수 내부에서 확인할 필요가 있다. 즉, 방어 코드가 필요하다.

JAVASCRIPT

```
function sum(x, y) {  
  // 인수가 전달되지 않아 매개변수의 값이 undefined인 경우, 기본값을 할당한다.  
  x = x || 0;  
  y = y || 0;  
  
  return x + y;  
}
```



```
}

```

```
console.log(sum(1, 2)); // 3
console.log(sum(1));    // 1

```

ES6에서는 **매개변수 기본값**을 사용하여 **함수 내에서 수행하던 인수 체크 및 초기화를 간소화할 수 있다.**

JAVASCRIPT

```
function sum(x = 0, y = 0) {
  return x + y;
}

console.log(sum(1, 2)); // 3
console.log(sum(1));    // 1

```

매개변수 기본값은 매개변수에 ^①인수를 전달하지 않았을 경우와 ^②undefined를 전달한 경우에만 유효하다.

JAVASCRIPT

```
function logName(name = 'Lee') {
  console.log(name);
}

logName();           // Lee
logName(undefined); // Lee
logName(null);       // null

```

앞서 살펴본 Rest 파라미터에는 기본값을 지정할 수 없다.

JAVASCRIPT

```
function foo(...rest = []) {
  console.log(rest);
}

// SyntaxError: Rest parameter may not have a default initializer

```

매개변수 기본값은 함수 정의 시 선언한 매개변수 개수를 나타내는 함수 객체의 length 프로퍼티와 arguments 객체에 영향을 주지 않는다.

JAVASCRIPT

```
function sum(x, y = 0) {  
  console.log(arguments);  
}  
  
console.log(sum.length); // 1  
  
sum(1);    // Arguments { '0': 1 }  
sum(1, 2); // Arguments { '0': 1, '1': 2 }
```