

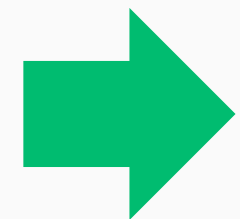
2.6.2 함수 ~

2.6.4 함수를 인수로 받는 배열 메서드

2.6.2 함수

함수

프로그래밍에서 특정한 작업을 수행하는 코드



★ 장점 ★

함수를 미리 만들어두면 원할 때 실행해 정해진 작업을 수행하게 할 수 있음.

SECTION 01

함수 선언하기

선언하다 : 함수를 만드는 행위 (변수도 동일)

방법 1.

function 예약어 사용

```
function() {};
```

방법 2.

화살표 기호(=>) 사용

```
() => {};
```

함수는 기본적으로 이름이 없음.(= 익명 함수)

문제점 → 다른 곳에서 사용 불가능

해결 방안 → 함수에 이름을 붙여야함.

** 익명 함수 : 이름 없는 함수

이름을 붙이는 방법

01

함수 선언문

함수를 상수에 대입하지 않고
function 뒤에 함수의 이름을
넣는 방식

02

함수 표현식

함수를 상수나 변수에 대입하
는 방식

03

화살표 함수

화살표 기호를 사용하는 함수

함수 선언문

함수를 상수에 대입하지 않고 function
뒤에 함수의 이름을 넣는 방식

형식

```
function 이름() { 실행문 }
```

EX:)

```
function a() {}
```


함수 표현식

함수를 상수나 변수에 대입하는 방식

형식

```
이름 = function() { 실행문 }
```

EX:)

```
const b = function() {};
```

화살표 함수

화살표 기호를 사용하는 함수

형식

```
() => { 실행문 }  
// 또는  
() => 반환식
```

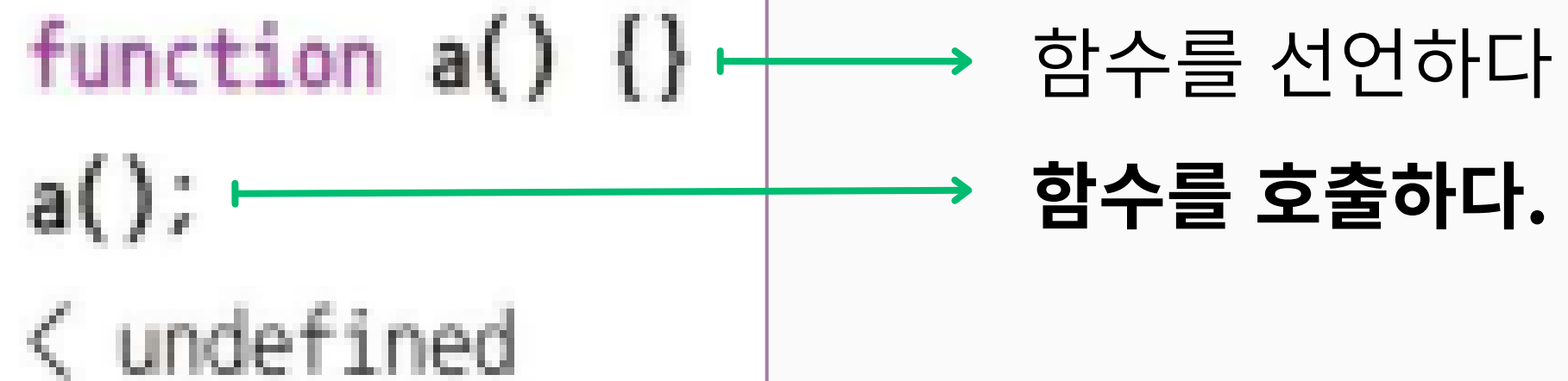
EX:)

```
const c = () => {};
```

SECTION 02

함수 호출하기

호출하다 : 함수를 사용하는 행위

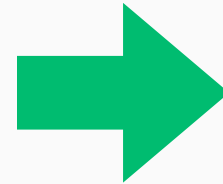


```
function a() {}  
a();  
< undefined
```

함수를 선언하다
함수를 호출하다.

** 아무 작업도 하지 않아 undefined 출력

EX:) 함수 안 실행문 넣기



★ 장점 ★

코드를 함수로 만들어 두면 코드를 재사용하기 좋음

```
function a() {  
  console.log('Hello');  
  console.log('Function');  
}  
a();  
Hello  
Function
```

```
a();  
a();  
a();  
Hello  
Function  
Hello  
Function  
Hello  
Function
```

SECTION 03

return 문으로 반환값 지정하기

반환값 : 함수를 호출하면 나오는 결과값

```
function a() {}  
a();  
< undefined
```

반환값

console.log()를 호출했을 때, undefined 출력
→ 이유는 ?
console.log() 함수의 반환값 = undefined

** 아무 작업도 하지 않아 undefined 출력

EX:) console.log() 함수의 반환값은 undefined

두 함수 의미 같음

```
function a() {  
  console.log('a');  
}
```

```
function a() {  
  console.log('a');  
  return undefined;  
}
```


return 문으로 반환값 지정하기

```
function a() {  
    return 10;  
}  
a();  
< 10
```

return문 뒤에 추가,
반환값을 직접 설정 가능

만약, return문을 사용하지 않았다면
항상 함수 뒤에 return undefined; 가 있다고 생각

return 문으로 반환값 지정하기

EX:) return 문 사용 X → 항상 함수 실행문 끝에 return undefined 존재

두 함수 의미 같음

```
function a() {}
```

```
function a() {  
    return undefined;  
}
```

★ 함수의 반환값 === 값 ★

- 함수의 반환값 상수나 변수에 대입
- 함수의 실행을 중간에 멈추는 역할

```
function a() {  
  return 10;  
}  
  
const b = a();  
  
b;  
< 10
```

```
function a() {  
  console.log('Hello');  
  return;  
  console.log('Return');  
}  
  
a();  
Hello
```

return 문으로 반환값 지정하기

- 함수의 실행을 중간에 멈추는 역할

EX:) 조건문과 return문을 결합해
함수 실행 조작하기

```
function a() {  
  if (false) {  
    return;  
  }  
  console.log(' 실행됩니다. ');  
}  
a();  
실행됩니다.
```

EX:) return 문으로 반복문 중단시키기

```
function b() {  
  for (let i = 0; i < 5; i++) {  
    if (i >= 3) {  
      return i;  
    }  
  }  
}  
b();  
< 3
```

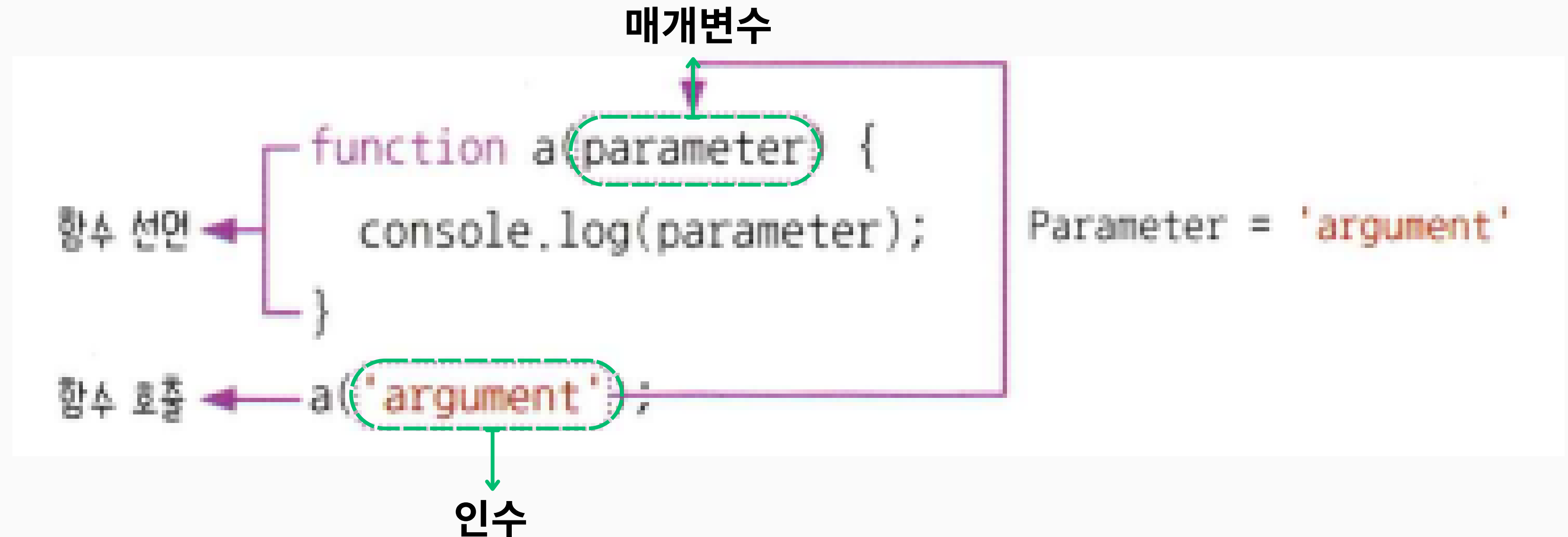
SECTION 04

매개변수와 인수 사용하기

- 매개변수와 인수의 관계

인수 : 함수를 호출할 때 넣은 값

매개변수 : 함수를 선언할 때 사용한 변수



- 매개변수 > 인수

```
function a(w, x, y, z) {  
  console.log(w, x, y, z);  
}  
a('Hello', 'Parameter', 'Argument');  
Hello Parameter Argument undefined
```

매개변수 개수 ≠ 인수 개수

→ 매개변수에 대응하는 인수가 없을 때는 자동으로 undefined 값이 대응

- 매개변수 < 인수

```
function a(w, x) {  
  console.log(w, x);  
}  
a('Hello', 'Parameter', 'Argument');
```

Hello Parameter

매개변수 개수 ≠ 인수 개수

→ 인수에 대응하는 매개변수가 없을 때,
대응하는 매개변수만 출력

- 인수의 개수 구하기

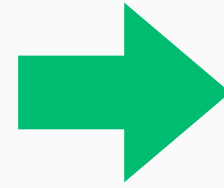
```
function a(w, x, y, z) {  
  console.log(w, x, y, z);  
  console.log(arguments);  
}  
a('Hello', 'Parameter', 'Argument');  
Hello Parameter Argument undefined  
(Arguments(3) ['Hello', 'Parameter', 'Argument'])
```

arguments 라는 값을 사용해
인수의 개수와 목록을 찾아낼 수 있음.

SECTION 05

다른 변수 사용하기

- 함수 안에서 변수나 상수 선언



순수 함수

자신의 매개변수나 내부 변수(=상수)만 사용하는 함수

```
function minus1(x, y) {  
  const a = 100;  
  return (x - y) * a;  
}
```

```
minus1(5, 3);  
< 200
```

상수 a

return 문에서 상수 a 사용

- 함수 바깥에 위치한 변수나 상수를 함수 안에서 사용

```
const a = 100;  
function minus2(x, y) {  
  return (x - y) * a;  
}  
minus2(5, 3);  
↵ 200
```

→ 함수를 선언하기 전,
상수 a를 이미 선언

→ 함수는 자신의 매개변수나 함수 내부에 선언한
변수 OR 상수가 아니더라도 접근 가능

→ 단, 모든 상수나 변수에 접근 불가능
스cope에 따라 접근 여부 달라짐

SECTION 06

고차 함수 사용하기

함수는 호출하면 어떤 값을 반환 (자바스크립트의 모든 자료형)

→ 함수가 함수도 반환 가능 !

```
const func = () => {  
  return () => {  
    console.log('hello');  
  };  
};
```

func() 함수를 호출하면 함수를 반환

- 함수를 호출하면 함수를 반환

EX:) 반환한 함수에 다른 변수 저장하고 변수에 저장된 함수를 다시 호출

```
const innerFunc = func();
```

→ 반환한 함수를 다른 변수에 저장

```
innerFunc();
```

→ 변수에 저장된 함수를 다시 호출

```
hello
```

- 함수 호출을 반환값으로 대체하기

```
const func = () => {  
  return () => {  
    console.log('hello');  
  };  
};
```

```
const innerFunc = func();
```

함수 호출 부분을
반환값으로 바꿔서
생각해 보기

```
const innerFunc = () => {  
  console.log('hello');  
};
```


- 함수를 호출할 때마다 다른 값으로 바꾸기

→ 반환값을 바꿀 때, 매개변수 사용하기 (바꾸고 싶은 자리를 매개변수로 만들기)

```
const func = (msg) => {  
  return () => {  
    console.log(msg);  
  };  
};
```

매개 변수

고차 함수 : 함수를 만드는 함수

```
const innerFunc1 = func('hello');  
const innerFunc2 = func('javascript');  
const innerFunc3 = func();  
innerFunc1();  
innerFunc2();  
innerFunc3();  
hello  
javascript  
undefined
```

실제 값

- 화살표 함수

→ 함수 본문에 바로 반환되는 값이 있으면 { 와 return 생략 가능

```
const func = (msg) => {  
  return () => {  
    console.log(msg);  
  };  
};
```

=

```
const func = (msg) => () => {  
  console.log(msg);  
};
```

화살표 함수가 연이어 나오기 가능 !

2.6.3 객체 리터럴

SECTION 01

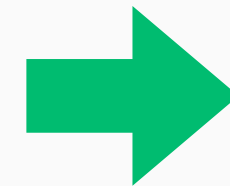
객체 생성하기

- 객체 생성하기 → 여러 변수에 저장된 정보를 하나로 묶기 가능

속성 : 종괄호로 묶인 정보

객체 리터럴 : 종괄호를 사용해 객체를 표현하는 것

```
const name = '조현영';  
const year = 1994;  
const month = 8;  
const date = 12;  
const gender = 'M';
```



```
const zerocho = {  
  name: '조현영',  
  year: 1994,  
  month: 8,  
  date: 12,  
  gender: 'M',  
};
```

속성 : 속성값

- 객체 구성

```
형식 {  
    <속성 이름>: <속성 값>,  
}
```

```
형식 {  
    <속성1 이름>: <속성1 값>,  
    <속성2 이름>: <속성2 값>,  
    <속성3 이름>: <속성3 값>,  
}
```

속성 이름 형태 → 문자열

속성 값 형태 → 자바스크립트 모든 값

속성이 여러 개 있다면 쉼표로 구분하기

SECTION 02

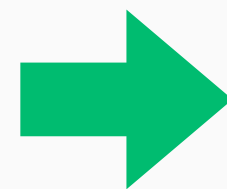
객체 속성에 접근하기

- 속성에 접근하기 = 속성 이름을 통해 속성 값에 접근하기

→ 속성에는 변수로 접근 가능

① 마침표(.)를 사용해 변수, 속성 형태로 접근

```
const zerocho = {  
  name: '조현영',  
  year: 1994,  
  month: 8,  
  date: 12,  
  gender: 'M',  
};
```



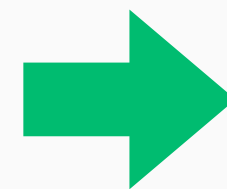
```
zerocho.name;  
< '조현영'
```


- 속성에 접근하기 = 속성 이름을 통해 속성 값에 접근하기

→ 속성에는 변수로 접근 가능

② 배열처럼 대괄호([])를 사용해 변수['속성'] 형태로 접근

```
const zerocho = {  
  name: '조현영',  
  year: 1994,  
  month: 8,  
  date: 12,  
  gender: 'M',  
};
```



```
zerocho['name'];  
// '조현영'
```

[] 에는 문자열에 넣어야 함으로 “ 필수 !

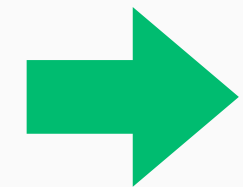
SECTION 03

객체 속성을 추가/수정/삭제하기

- 값 추가 & 수정

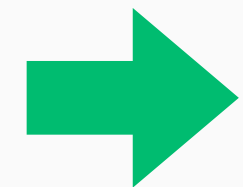
형태 : 변수.속성 = 값;

```
zerocho.married = false; // false  
zerocho.married; // false
```



추가

```
zerocho.gender = 'F'; // F
```




수정

- 값 삭제

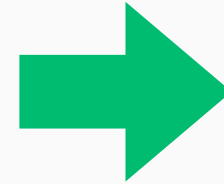
형태 : `delete 변수.속성;`

```
delete zerocho.gender; // true  
zerocho.gender; // undefined
```



삭제한 속성에 접근하면
undefined가 나옴.

- 배열과 함수가 객체인 이유?



객체의 성질을 모두 다 사용할 수 있기 때문.

함수에 속성 추가하기

```
function hello() {}  
hello.a = 'really?';  
hello.a;  
< 'really?'
```

배열에 속성 추가하기

```
const array = [];  
array.b = 'wow!';  
array;  
< [b: 'wow!']  
array.b;  
< 'wow!'
```

- 객체 \supset 함수 & 배열

→ 중괄호로 만든 객체를 구분하기 위해 **객체 리터럴**이라고 따로 지칭

다만, 함수와 배열은 주로 객체 리터럴과 다른 목적으로 사용함.

→ 함수와 배열에 임의 속성을 추가하는 경우는 드뭄.

- **목적 차이**

객체 리터럴 → 속성(key-value)을 정적으로 저장/조회하려는 목적

배열 → 순서 있는 값 집합

▶ 주로 index를 기준으로 반복 처리

함수 → 실행 가능한 로직 단위

▶ 코드 재사용이나 이벤트 처리를 위해 정의

SECTION 04

메서드 이해하기

- 메서드 : 객체의 속성 값으로 함수가 들어가는 속성

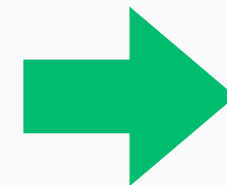
```
const debug = {  
  log: function(value) {  
    console.log(value);  
  },  
};  
debug.log('Hello, Method');  
Hello, Method
```

log = debug의 속성

log의 속성값 = 함수
→ 메서드

- 메서드 : 객체의 속성 값으로 함수가 들어가는 속성

```
const debug = {  
  log: function(value) {  
    console.log(value);  
  },  
};  
debug.log('Hello, Method');  
Hello, Method
```



```
const debug = {  
  log(value) {  
    console.log(value);  
  },  
};
```

SECTION 05

객체 간 비교하기

- **ture** 경우

숫자, 문자열, 불 값, null, undefined 비교

```
'str' === 'str'; // true  
123 === 123; // true  
false === false; // true  
null === null; // true  
undefined === undefined; // true
```

- **false** 경우

NaN, 객체 비교

```
NaN === NaN; // false
```

```
(({} === {}));  
< false
```


객체는 모양이 같아도 생성할 때마다
새로운 객체가 생성

SECTION 06

중첩된 객체와 옵셔널 체이닝 연산자

- 객체 안 다른 객체 = 중첩된 객체

```
const zerocho = {  
  name: {  
    first: '현영',  
    last: '조',  
  },  
  gender: 'm',  
};
```



zerocho.name.first
zerocho ['name'] ['first']
zerocho ['name'].first
zerocho.name['first']

- 배열 안 객체 & 객체 안 배열

```
const family = [  
  { name: '제로초', age: 29, gender: '남' },  
  { name: '레오', age: 5, gender: '남' },  
  { name: '체리', age: 3, gender: '여' },  
];
```


- 옵셔널 체이닝 연산자

→ 존재하지 않는 속성에 접근할 때, 에러가 발생하는 것을 막아줌.

```
zerocho.girlfriend?.name;
```

```
< undefined
```

```
zerocho.name?.first;
```

```
< '현영'
```

없는 속성에 접근할 때,

→ 그 안에 name 속성에 접근하는 것을 막고 undefined를 결과로 보냄.

- **옵셔널 체이닝 연산자**

→ 존재하지 않는 속성에 접근할 때, 에러가 발생하는 것을 막아줌.

```
zerocho.sayHello?.();  
< undefined  
zerocho.girlfriend?.[0];  
< undefined
```

메서드나 배열 요소에 접근할 때도
?. 연산자 사용 가능

SECTION 07

참조와 복사

객체를 저장한 변수를 다른 변수에 대입

→ **참조**

객체가 아닌 값(문자열, 숫자, 불 값, null, undefined)에 다른 변수를 대입

→ **복사 (참조 관계가 생기지 않음)**

• 참조

```
const a = { name: 'zerocho' };
```

```
const b = a;
```

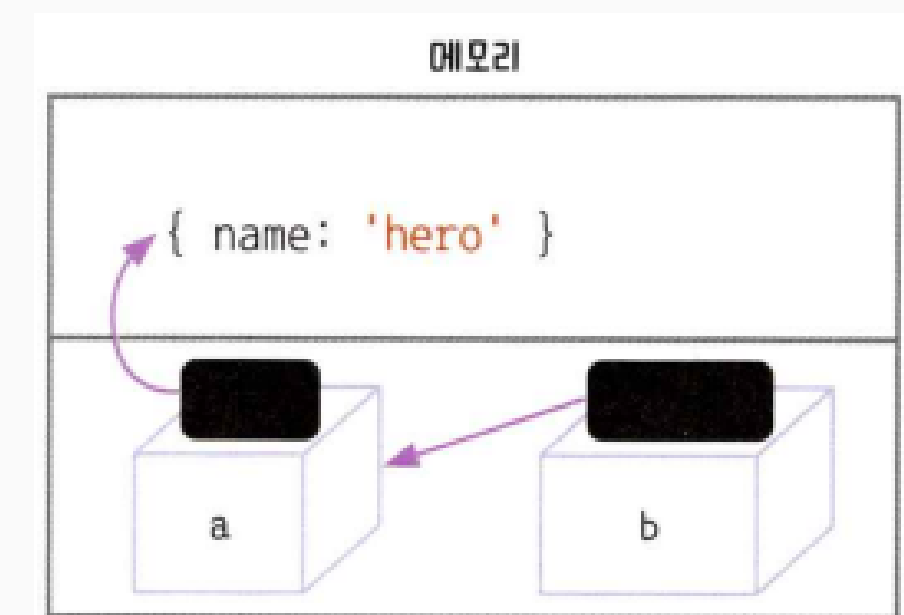
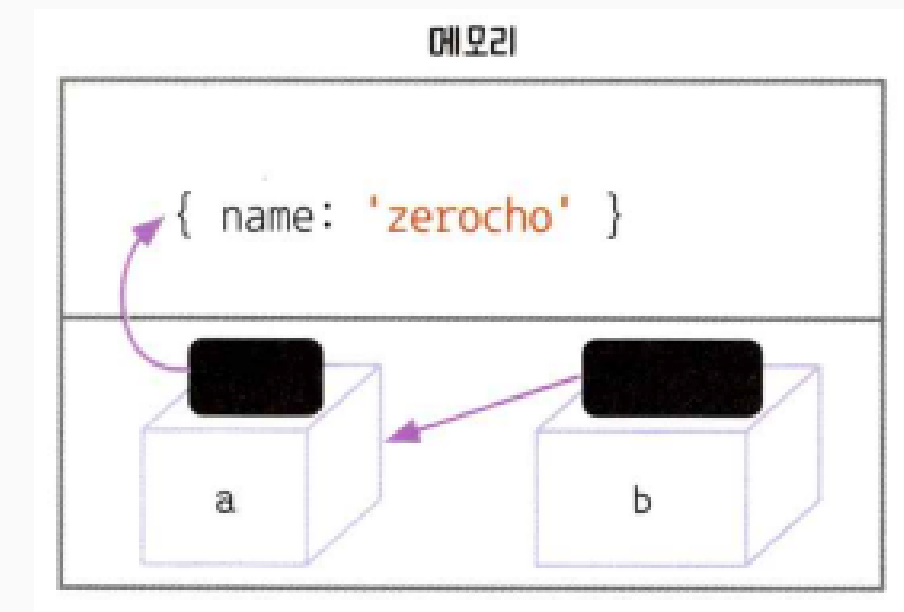
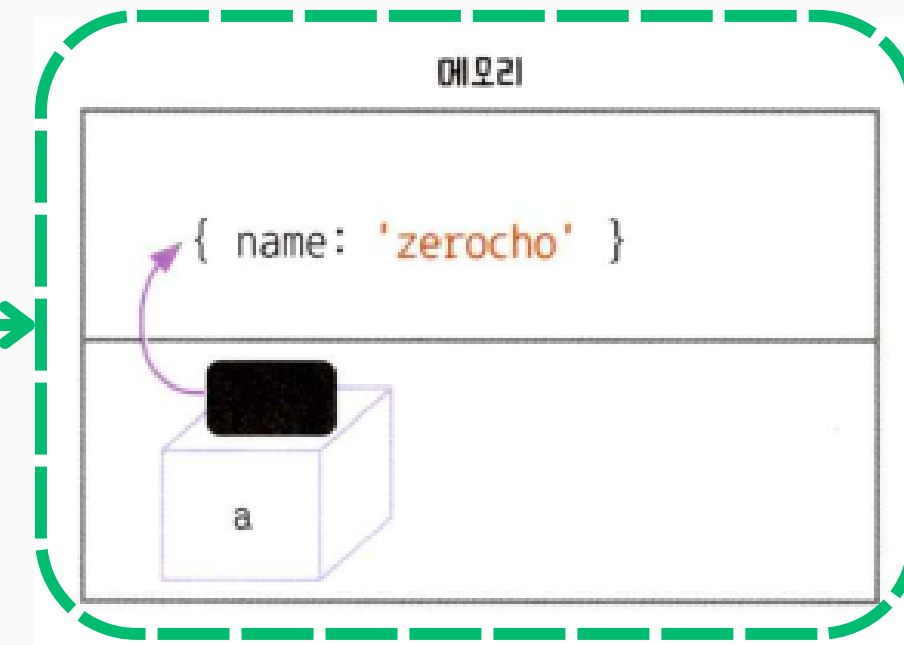
```
b.name;
```

```
< 'zerocho'
```

```
a.name = 'hero';
```

```
b.name;
```

```
< 'hero'
```



• 참조

```
const a = { name: 'zerocho' };
```

```
const b = a;
```

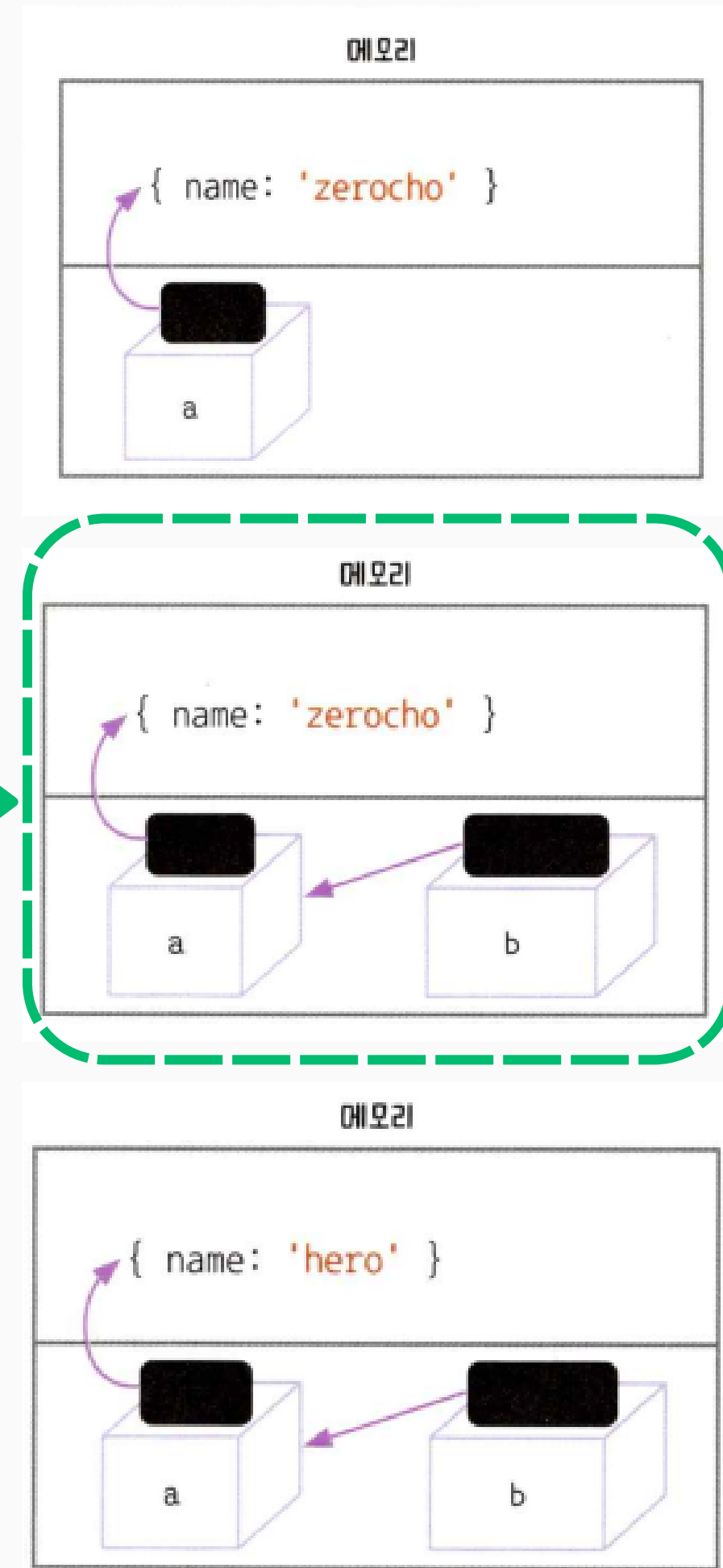
```
b.name;
```

```
< 'zerocho'
```

```
a.name = 'hero';
```

```
b.name;
```

```
< 'hero'
```



• 참조

```
const a = { name: 'zerocho' };
```

```
const b = a;
```

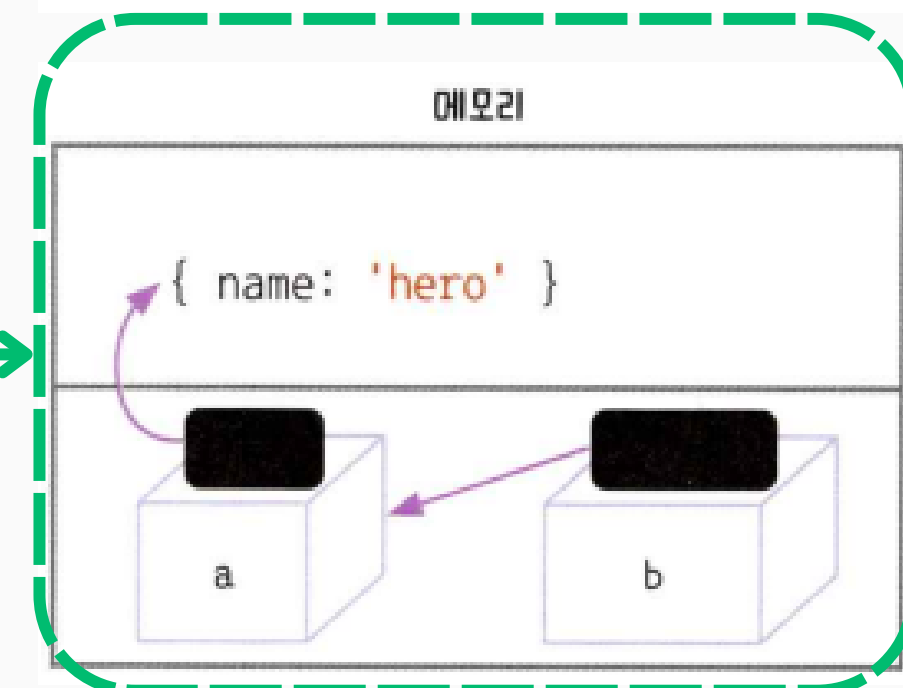
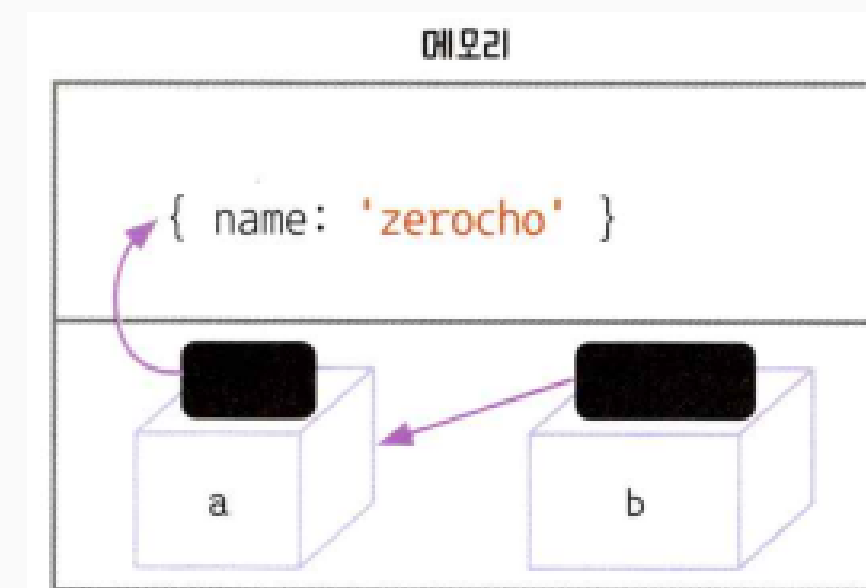
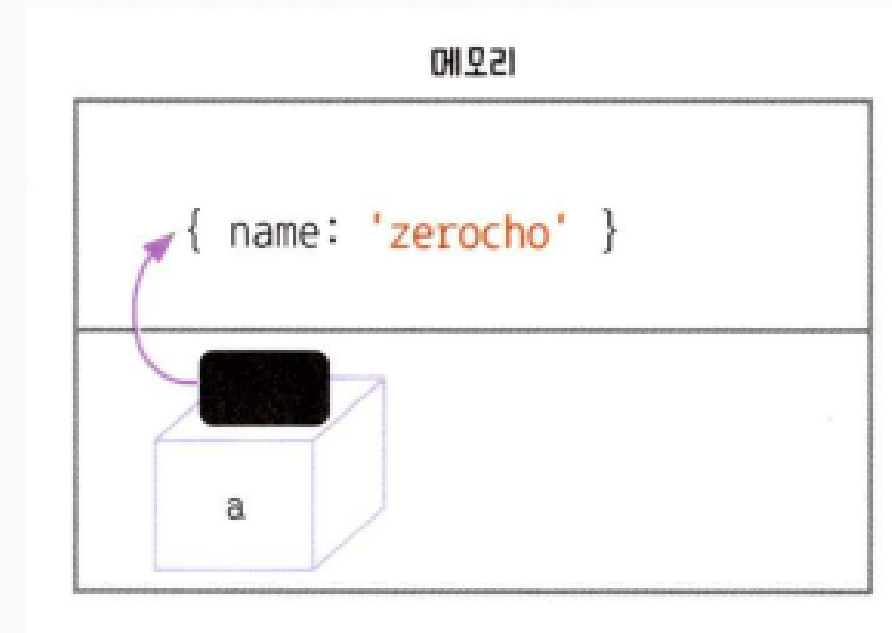
```
b.name;
```

```
< 'zerocho'
```

```
a.name = 'hero';
```

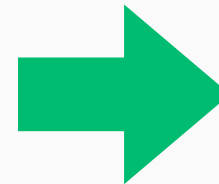
```
b.name;
```

```
< 'hero'
```



- 참조

```
const a = { name: 'zerocho' };  
const b = a;  
b.name;  
< 'zerocho'  
a.name = 'hero';  
b.name;  
< 'hero'
```

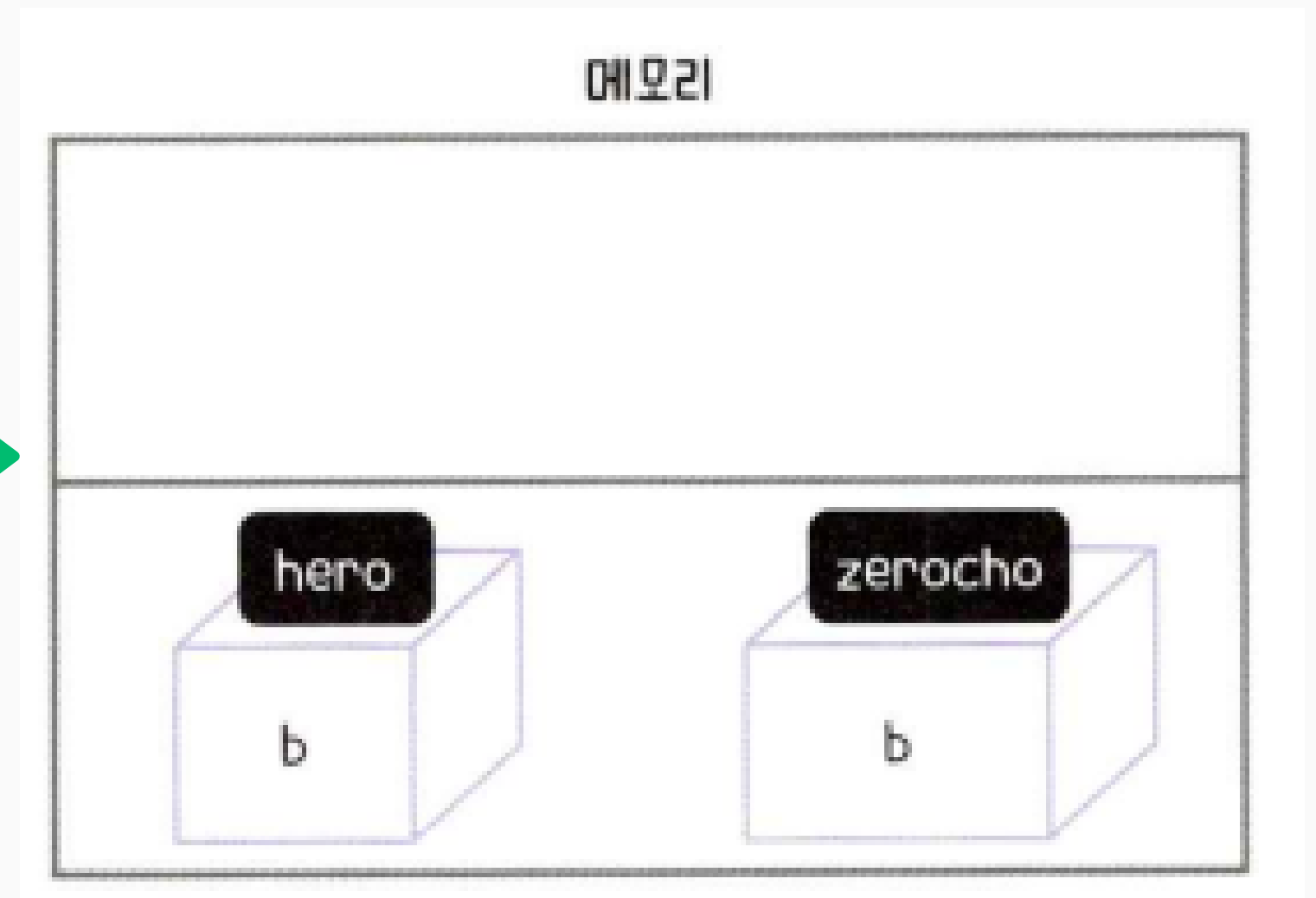


a와 b가 같은 객체를 참조하고 있다.
변수와 a와 b 그리고 객체 간에 참조 관계가 있다.

- 복사

= 참조관계가 끊기는 것

```
let a = 'zerocho';  
let b = a;  
a = 'hero';  
b;  
< 'zerocho'
```

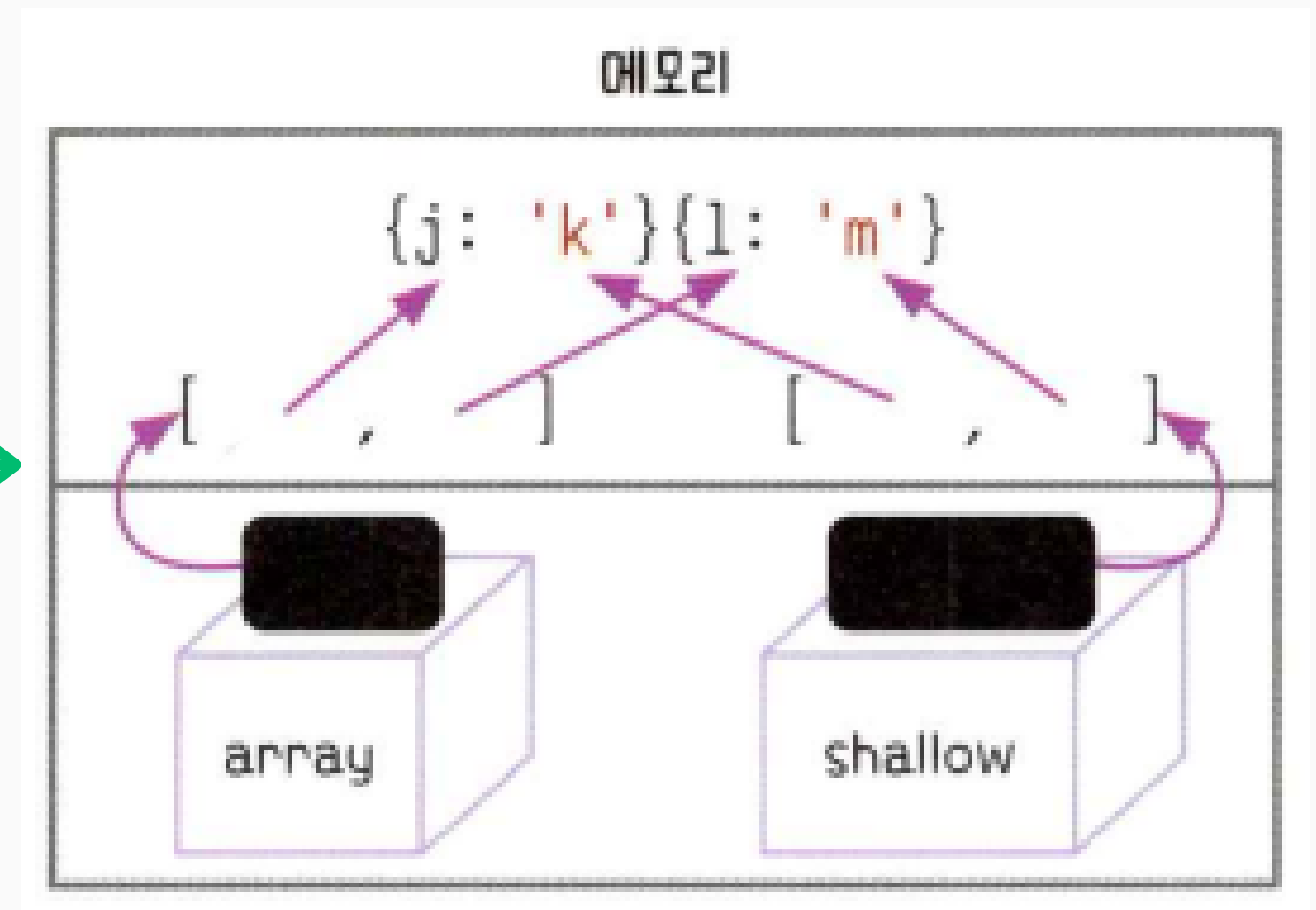


참조와 복사

- 얇은 복사 : 외부 객체만 복사 되고 내부 객체는 참조 관계를 유지하는 복사

```
const array = [{ j: 'k' }, { l: 'm' }];  
const shallow = [...array]; // 얇은 복사  
console.log(array === shallow); // false  
console.log(array[0] === shallow[0]); // true
```

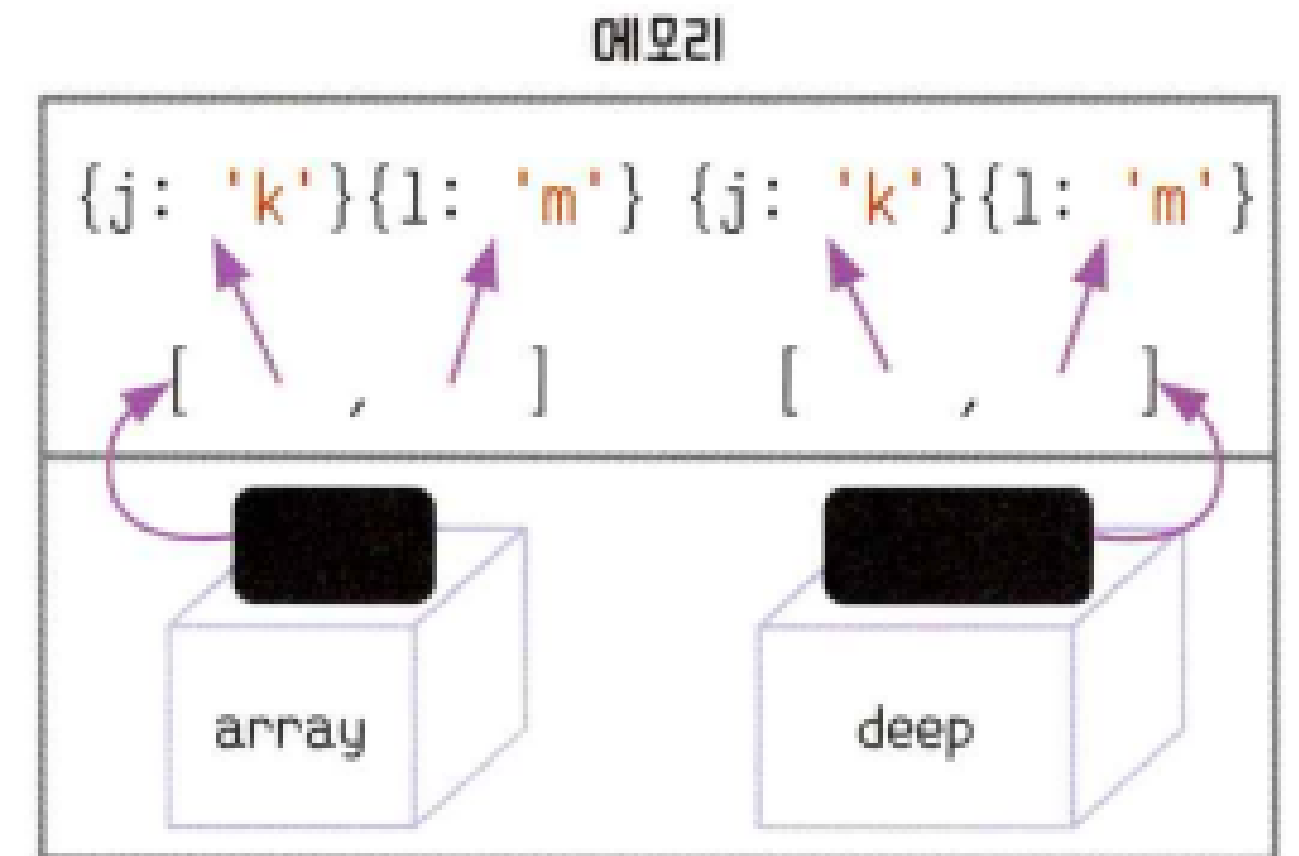
얇은 복사를 할 때는
... 연산자 사용
(=스프레드 문법)



- **깊은 복사** : 내부객체까지 참조 관계가 끊기면서 복사

```
const array = [{ j: 'k' }, { l: 'm' }];  
const deep = JSON.parse(JSON.stringify(array)); // 깊은 복사  
console.log(array === deep); // false  
console.log(array[0] === deep[0]); // false
```

JSON.parse() : 문자열 → 객체
JSON.stringify() : 객체 → 문자열



SECTION 08

구조 분해 할당

- 구조 분해 할당 : 객체에서 객체의 속성 이름과 대입하는 변수명이 같을 때 줄여서 쓰는 문법

```
let a = 5;
```

```
let b = 3;
```

```
let temp = a;
```

```
a = b;
```

```
b = temp;
```

→ 구조 분해 할당 문법이 나오기 전 방법

```
const obj = { a: 1, b: 2 };
```

```
const a = obj.a;
```

```
const b = obj.b;
```

```
const { a, b } = obj; // 앞의 두 줄을 이렇게 한 줄로 표현 가능
```

```
a; // 1
```

→ 구조 분해 할당 문법

구조 분해 할당

```
const array = [1, 2, 5];  
const one = array[0];  
const two = array[1];  
const five = array[2];  
const [one, two, five] = array; // 앞의 세 줄을 이렇게 한 줄로 표현 가능  
two; // 2
```

→ 배열

```
let a = 5;  
let b = 3;  
[b, a] = [a, b]; // (2) [5, 3]
```

→ 선언된 변수

SECTION 09

유사배열 객체

- 유사 배열 객체 : 배열 모양을 한 객체

배열 메서드 사용하기 위해,
Array.from() 메서드로 유사 배열 객체를 배열로 변경

```
const array = {  
  0: 'hello',  
  1: 'I\'m',  
  2: 'Object',  
  length: 3,  
}  
array[0]; // 'hello'  
array[1]; // "I'm"  
array[2]; // 'Object'  
array.length; // 3  
array.push(1); // Uncaught TypeError: array.push is not a function
```

배열이 아니므로 배열 메서드 사용 할 수 없음.

```
Array.from(array).indexOf('hello'); // 0
```


2.6.4 함수를 인수로 받는 배열 메서드

SECTION 01

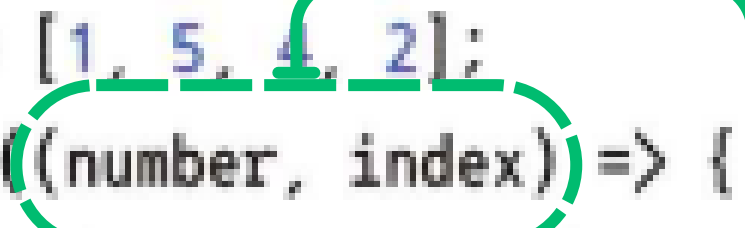
forEach()와 map()

forEach()와 map()

- **forEach** : 각 배열 요소에 대해 제공된 함수를 한 번씩 실행

형식 배열.forEach(함수);

```
const arr = [1, 5, 4, 2];  
arr.forEach((number, index) => {  
  console.log(number, index);  
});  
1 0  
5 1  
4 2  
2 3
```



인수로 받은 함수의 매개 변수

: 요소 (number), 요소의 인덱스(index)

- ▶ 이름은 마음대로 지을 수 있으며 첫 번째 매개변수가 배열의 요소, 두 번째가 요소의 인덱스인건 변하지 않음.
- ▶ 사용하지 않는 매개변수는 생략 가능하지만, 요소 인덱스를 사용하는 경우에는 배열의 요소를 가리키는 매개 변수를 사용해야함.

콜백 함수

: 다른 메서드에 인수로 넣을 때 실행되는 함수

forEach()와 map()

- **map()** : 호출한 배열의 모든 요소에 주어진 함수를 호출한 결과로 채운 새로운 배열을 생성

형식 배열.map(<콜백 함수>);

```
const numbers = [];  
for (let n = 1; n <= 5; n += 1) {  
  numbers.push(n); // 1부터 5까지의 배열  
}  
numbers; // (5) [1, 2, 3, 4, 5]
```

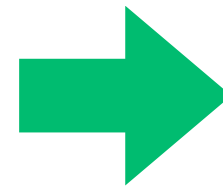


그림 2-35 map() 메서드의 작동 방식

[1, 1, 1, 1, 1]

map((v, i) => i + 1)

[1, 2, 3, 4, 5]

v	1	1	1	1	1
i	0	1	2	3	4
i + 1	1	2	3	4	5

```
const numbers = Array(5).fill(1).map((v, i) => i + 1);  
numbers; // (5) [1, 2, 3, 4, 5]
```

forEach()와 map()

- **map()** : 호출한 배열의 모든 요소에 주어진 함수를 호출한 결과로 채운 새로운 배열을 생성

```
const array = [1, 3, 5, 7];  
const newArray = array.map((v, i) => {  
  return v * 2;  
});  
console.log(array); // [1, 3, 5, 7]  
console.log(newArray); // [2, 6, 10, 14]
```

map()은 항상 새로운 배열을 반환하므로
반환한 배열을 변수에 저장해야 함.
→ **원본 배열은 수정되지 않음.**

SECTION 02

find(), findIndex(), filter()

find(), findIndex(), filter()

- **find()** : 콜백 함수의 반환값이 true인 요소를 찾는 메서드

```
배열.find(<콜백 함수>);
```

```
const array = [1, 3, 5, 7];  
array.find((v, i) => {  
  return v > 1;  
});  
< 3
```

true인 요소가 여러 개 일 경우에는 **처음 찾은 요소**를 반환

EX:) 3, 5, 7은 조건에 맞지만 3 > 1 은 true이니 3을 반환 하며,
true를 찾았으니 5와 7은 검사하지 않음

find(), findIndex(), filter()

- **find()** : 콜백 함수의 반환값이 true인 요소를 찾는 메서드

```
const nested = [{ age: 29 }, { age: 5 }, { age: 3 }];  
nested.includes({ age: 29 }); // false
```

```
const nested = [{ age: 29 }, { age: 5 }, { age: 3 }];  
nested.find((v) => v.age === 29); // { age: 29 }
```

{ age : 29 } !== { age : 29 } 이기 때문에
이럴 땐, find()로 찾는 것이 나옴.

find(), findIndex(), filter()

- **findIndex()** : 찾은 요소의 인덱스를 반환하고, 찾지 못했다면 -1을 반환

```
const array = [1, 3, 5, 7];  
array.findIndex((v, i) => {  
  return v > 1;  
}); // 1
```

find(), findIndex(), filter()

- **filter()** : **find()**처럼 콜백 함수의 반환값이 **true**가 되는 요소를 찾지만, 하나만 찾는 것이 아니라 해당하는 요소를 찾아 배열로 반환

→ 배열 내부에 객체가 있는 경우, 조건에 해당하는 속성을 모두 찾으려면 filter 사용 !

```
const array = [1, 3, 5, 7];  
array.filter((v, i) => {  
  return v > 1;  
}); // [3, 5, 7]
```

```
const nested = [{ age: 29 }, { age: 5 }, { age: 3 }];  
nested.filter((v) => v.age < 29); // (2) [{ age: 5 }, { age: 3 }]
```

SECTION 03

sort()

sort()

- **sort() : 비교 함수의 반환값에 따라 배열을 정렬하는 메서드**

형식 배열.sort(<비교 함수>):

- 배열 요소를 기본적으로 문자열 기준으로 정렬
- 숫자를 원하는 기준으로 정렬하려면 비교 함수 사용

sort()

- **sort() : 비교 함수의 반환값에 따라 배열을 정렬하는 메서드**

형식

(a, b) => 반환값

음수 반환 → a가 앞

양수 반환 → b가 앞

0 반환 → 순서 그대로

▶ 배열의 모든 요소에 대해 반복적으로 비교함

```
const arr = [1, 5, 4, 2, 3];  
arr.sort((a, b) => a - b);  
arr; // (5) [1, 2, 3, 4, 5];
```

```
const arr = [1, 5, 4, 2, 3];  
arr.sort((a, b) => b - a);  
arr; // [5, 4, 3, 2, 1];
```

```
const arr = [1, 5, 4, 2, 3];  
const shallow = [...arr];  
shallow.sort((a, b) => b - a);  
arr; // [1, 5, 4, 2, 3];  
shallow; // [5, 4, 3, 2, 1];
```

SECTION 04

reduce()

reduce()

- **reduce()** : 배열의 각 요소를 하나의 값으로 누적해서 축약하는 메서드

```
배열.reduce((〈누적 값〉, 〈현재 값〉) => {  
  return 〈새로운 누적 값〉;  
}, 〈초기 값〉);
```

```
[1, 2, 3, 4, 5].reduce((a, c) => {  
  return a + c;  
}, 0); // 15
```

a(누적 값)	0	1	3	6	10
c(현재 값)	1	2	3	4	5
a + c(반환값)	1	3	5	10	15

reduce()

- **reduce()** : 배열의 각 요소를 하나의 값으로 누적해서 축약하는 메서드

```
배열.reduce((〈누적 값〉, 〈현재 값〉) => {  
  return 〈새로운 누적 값〉;  
}, 〈초기 값〉);
```

초기 값을 제공하지 않으면
첫 번째 요소 값이 초기 값이 됨.

```
[1, 2, 3, 4, 5].reduce((a, c) => {  
  return a + c;  
}); // 15
```

a(누적 값)	1	3	6	10
c(현재 값)	2	3	4	5
a + c(반환값)	3	6	10	15

SECTION 03

every()와 some()

every()와 some()

- **every()** : 모든 요소가 조건을 만족해야 true

배열.every(<조건 함수>);

```
const array = [1, 3, 5, 7]  
array.every((value) => value !== null); // true
```

→ 모든 요소가 null이 아니므로 true

```
const array = [1, 3, 5, null, 7];  
array.every((v) => v !== null); // false
```

→ 하나라도 null이면 false

every()와 some()

- **some()** : 하나라도 조건을 만족하면 true

```
배열.some(<조건 함수>);
```

```
const array = [1, 3, null, 7];  
array.some((v) => v === null); // true
```

→ null 하나라도 있으면 true

```
const array = [1, 3, 5, 7];  
array.some((v) => v === null); // false
```

→ 조건에 맞는 요소가 하나도 없으면 false