Chap27 배열

▼ 배열이란?

- **배열** : 여러 개의 값을 순차적으로 나열한 자료구조
- **요소** : 배열이 가직 있는 값
- **인덱스** : 배열에서 자신의 위치를 나타내는 0이상의 정수. 인덱스를 통해 배열의 요소에 접근한다.
- **length 프로퍼티** : 배열 요소의 개수, 길이를 나타냄

```
const arr=['apple','banana','arange'];
cnsole.log(arr.length); //3
```

- 배열의 순회 : for문 사용
- 배열의 타입 : obejct
- 배열 생성방법 ： 배열 리터럴, Array 생성자 함수, Array.of, Array.from
- 배열과 객체의 차이

구분	객체	배열
구조	프로퍼티 키와 프로퍼티 값	인덱스와 요소
값의 참조	프로퍼티 키	인덱스
값의 순서	X	O
length 프로퍼티	X	O

▼ 자바스크립트 배열은 배열이 아니다

자료구조에서 말하는 배열은 동일한 크기의 메모리 공간이 **빈틈없이 연속적으로 나열된 자료구조** 즉 **“밀집 배열”**을 뜻하지만

자바스크립트에서 배열은 요소를 위한 각각의 메모리 공간이 동일한 크기를 갖지 않아도 되며, 연속적으로 이어져 있지 않을 수도 있는 **희소배열**이다.

▼ 일반적인 배열과 자바스크립트 배열의 장단점?

- **일반적인 배열**은 인덱스로 요소에 빠르게 접근할 수 있다. 하지만 요소를 삽입 또는 삭제하는 경우에는 효율적이지 않다.
- **자바스크립트 배열**은 해시 테이블로 구현된 객체이므로 인덱스로 요소에 접근하는 경우 일반적인 배려보다 성능적인 면에서 느릴 수 밖에 없는 구조적인 단점이 있다. 하지만 요소를 삽입 또는 삭제하는 경우에는 일반적인 배열보다 빠른 성능을 기대할 수 있다.

▼ length 프로퍼티와 희소 배열

희소배열은 length와 배열 요소의 개수가 일치하지 않는다. 희소 배열의 length는 희소 배열의 실제 요소 개수보다 언제나 크다.

따라서 배열을 생성할 경우에는 희소배열 대신 요소를 연속적으로 위치시키는 것이 좋다.

▼ 배열 생성

- 배열 리터럴
- Array 생성자 함수
- Array.of
- Array.from

```
//배열 리터럴
let arr = [1, 2, 3, 4, 5];
console.log(arr); // [1, 2, 3, 4, 5]

//Array 생성자 함수
let arr = new Array(1, 2, 3, 4, 5);
console.log(arr); // [1, 2, 3, 4, 5]

//Array.of
let arr = Array.of(1, 2, 3, 4, 5);
console.log(arr); // [1, 2, 3, 4, 5]
```

```
//Array.from
let arr = Array.from('hello');
console.log(arr); // ['h', 'e', 'l', 'l', 'o']
```

▼ 배열 요소의 참조

- 대괄호([]) 표기법 사용
- 존재하지 않는 요소에 접근하면 undefiend가 반환된다.
- 희소배열의 존재하지 않는 요소를 차조해도 undefiend가 반환된다.

```
//희소배열 arr
const arr=[1,,3];
console.log(arr[1]);//undefined
console.log(arr[3]);//undefined
```

▼ 배열 요소의 추가와 갱신

- 인덱스를 사용해 값을 할당하면 새로운 요소가 추가된다.
- 현재 배열의 length프로퍼티 값보다 큰 인데스로 새로운 요소를 추가하면 희소 배열이 된다.
- 인덱스로 요소에 접근하여 명시적으로 값을 할당하지 않은 요소는 생성되지 않는다.
- 이미 요소가 존재하는 요소에 값을 재할당하면 요소값이 갱신된다.
- 만약 **정수(또는 정수 형태의 문자열) 이외의 값**을 인덱스처럼 사용하면 요소가 생성되는 것이 아니라 **프로퍼티가 생성**된다.

```
const arr=[];
arr[0]=1;
console.log(arr); //[1]

arr['1']=2;
console.log(arr); //[1,2]

arr['foo']=3;
console.log(arr);//[1,2,foo:3]
```

▼ 배열 요소의 삭제

- 배열은 객체이기 때문에 `delete연산자`를 사용하여 배열 요소를 삭제할 수 있다. 단, **희소배열을 생성**할 수 있다.

```
const arr=[1.2,3];
delete arr[1];
//희소배열이 되어버림
console.log(arr);//[1,empty,3]
console.log(arr.length);//3
```

- Array.prototype.splice 메서드 사용 (희소배열 생성 X)

```
const arr=[1.2,3];
//Array.prototype.splice(삭제를 시작할 인덱스, 삭제할 요소 수)
arr.splice(1,1);
console.log(arr);//[1,3]
console.log(arr.length);//2
```

▼ 배열 메서드

- 원본 배열을 직접 변경하는 메서드와 원본 배열을 직접 변경하지 않고 새로운 배열을 생성하여 반환하는 메서드로 나뉘므로 사용할 대 주의해야한다.
- 가급적 원본 배열을 직접 변경하지 않는 메서드를사용하는 편이 좋다.

▼ Array.isArray

- Array생성자 함수의 정적 메서드
- 전달된 인수가 배열이면 true, 아니면 false

```
// true
Array.isArray([]);
Array.isArray([1, 2]);
Array.isArray(new Array());

// false
Array.isArray();
Array.isArray({});
Array.isArray(null);
```

```
Array.isArray(undefined);
Array.isArray(1);
Array.isArray('Array');
Array.isArray(true);
Array.isArray(false);
Array.isArray({ 0: 1, length: 1 })
```

▼ Array.prototype.indexOf

- 인수로 전달된 요소를 검색하여 인덱스를 반환한다.
- 따라서 특정 요소가 존재하는지 확인할 때 유용
- 만약 검색한 요소가 여러개 있다면 첫번째로 검색된 요소의 인덱스를 반환한다.
- 검색한 요소가 없다면 -1을 반환한다.

▼ Array.prototype.push

- 인수로 전달받은 모든 값을 원본 배열의 **마지막 요소로 추가**하고 변경된 **length 프로퍼티 값을 반환**
- 원본 배열을 직접 변경
- 성능면에서 좋지 않고 부수효과 있어 스프레드 문법을 사용하는 것이 더 좋다.
- 큐, 스택 구현에 유용

▼ 문제

다음 반환할 값은?

```
const arr = [1, 2];

let result = arr.push(3, 4);
console.log(result); //1번

// push 메서드는 원본 배열을 직접 변경한다.
console.log(arr);2번
```

▼ 답

1번 : 4

2번 : [1,2,3,4]

▼ Array.prototype.pop

- 마지막 요소를 제거하고 제거한 요소를 반환
- 원본 배열이 빈 배열이면 undefiend를 반환
- 원본 배열 직접 변경
- 스택 구현에 유용

```
const Stack = (function () {
  function Stack(array = []) {
    if (!Array.isArray(array)) {
      // "47. 에러 처리" 참고
      throw new TypeError(`${array} is not an array.`
    }
    this.array = array;
  }

  Stack.prototype = {
    // "19.10.1. 생성자 함수에 의한 프로토타입의 교체" 참고
    constructor: Stack,
    // 스택의 가장 마지막에 데이터를 밀어 넣는다.
    push(value) {
      return this.array.push(value);
    },
    // 스택의 가장 마지막 데이터, 즉 가장 나중에 밀어 넣은 최신
    pop() {
      return this.array.pop();
    },
    // 스택의 복사본 배열을 반환한다.
    entries() {
      return [...this.array];
    }
  };

  return Stack;
}());

const stack = new Stack([1, 2]);
console.log(stack.entries()); // [1, 2]
```

```
stack.push(3);
console.log(stack.entries()); // [1, 2, 3]

stack.pop();
console.log(stack.entries()); // [1, 2]
```

▼ 문제

```
const arr = [1, 2];

let result = arr.pop();
console.log(result); //1번

console.log(arr); // 2번
```

▼ 답

1번 : 2 / 제거한 요소를 반환하기 때문에 2를 반환

2번 : [1]

▼ Array.prototype.unshift

- 인수로 전달받은 모든 값을 **원본 배열의 선두에 요소로 추가**하고 변경된 length 프로퍼티 값을 반환
- 원본 배열을 직접 변경
- 부수효과 있기때문에 스프레드 문법을 사용하는 것이 좋다.

▼ 문제

```
const arr = [1, 2];

let result = arr.unshift(3, 4);
console.log(result); // 1번
console.log(arr); // 2번
```

▼ 답

1번 : 4 / 추가되고 난 후 배열의 길이를 반환하기 때문에 4

2번 : [3,4,1,2] / 선두에 추가하기 때문에 3,4는 앞으로!

▼ Array.prototype.shift

- 첫번째 요소를 제거하고 제거한 요소를 반환.
- 빈배열이면 undefiend반환
- 원본 배열 직접 변경
- 큐(처음 들어간것이 처음으로 나온다. 선입 선출) 구현에 유용

```
const Queue = (function () {
  function Queue(array = []) {
    if (!Array.isArray(array)) {
      // "47. 에러 처리" 참고
      throw new TypeError(`${array} is not an array.`
    }
    this.array = array;
  }

  Queue.prototype = {
    // "19.10.1. 생성자 함수에 의한 프로토타입의 교체" 참고
    constructor: Queue,
    // 큐의 가장 마지막에 데이터를 밀어 넣는다.
    enqueue(value) {
      return this.array.push(value);
    },
    // 큐의 가장 처음 데이터, 즉 가장 먼저 밀어 넣은 데이터를
    dequeue() {
      return this.array.shift();
    },
    // 큐의 복사본 배열을 반환한다.
    entries() {
      return [...this.array];
    }
  };

  return Queue;
}());
```

```
const queue = new Queue([1, 2]);
console.log(queue.entries()); // [1, 2]

queue.enqueue(3);
console.log(queue.entries()); // [1, 2, 3]

queue.dequeue();
console.log(queue.entries()); // [2, 3]
```

▼ 문제

```
const arr = [1, 2];

let result = arr.shift();
console.log(result); //1번

console.log(arr); //2번
```

▼ 답

1번 : 1 / 첫번째 요소 제거 후 제거한 요소 반환

2번 : [2]

▼ Array.prototype.concat

- 인수로 전달된 값들을 원본 배열의 **마지막 요소**로 추가한 **새로운 배열로 반환(원본 배열 변경X)**
- 인수로 전달한 값이 배열인 경우 배열을 **해체하여 새로운 배열의 요소**로 추가한다.

  (**unshift와 push 메서드**는 인수로 **전달받은 배열을 그대로 원본 배열의 요소로 추가**한다)

```
const arr = [3, 4];

// unshift와 push 메서드는 인수로 전달받은 배열을 그대로 원본
arr.unshift([1, 2]);
arr.push([5, 6]);
console.log(arr); // [[1, 2], 3, 4,[5, 6]]
```

```
// concat 메서드는 인수로 전달받은 배열을 해체하여 새로운 배열의
let result = [1, 2].concat([3, 4]);
result = result.concat([5, 6]);

console.log(result); // [1, 2, 3, 4, 5, 6]
```

▼ 문제

```
const arr1 = [1, 2];
const arr2 = [3, 4];

let result = arr1.concat(arr2);
console.log(result); //1번

result = arr1.concat(3);
console.log(result); //2번


result = arr1.concat(arr2, 5);
console.log(result); //3번

console.log(arr1); // 4번

result=arr1.concat([5,6])
console.log(result); //5qjs
```

▼ 답

1번 : [1,2,3,4] / 마지막 요소로 추가하니깐 뒤에 넣어야함

2번 : [1,2,3] / 원본 배열 훼손 안하니깐 1,2,3이 된다.

3번 : [1,2,3,4,5]

4번 : [1,2] / 원본 배열 훼손하지 않음

5번 : [1,2,5,6] / 배열을 추가하면 해체하여 요소로 추가

▼ Array.prototype.splice

- 원본 배열의 **중간에 요소**를 추가하거나 중간에 있는 요소를 제거하는 경우에 사용

- 원본 배열을 직접 변경
- 형식

```
//제거할 요소 개수가 0으로 지정하면 아무런 요소도 제거하지 않고
//삽입할 요소들을 적지 않으면 제거만 하면 된다.
//제거할 요소의 개수를 생략하면 시작인덱스부터 모두 제거
//반환값 : 제거한 요소기 배열로 반환
arr.slice(시작인덱스,제거할 요소의 개수,삽입할 요소들~)
```

▼ 문제

```
const arr = [1, 2, 3, 4];

const result = arr.splice(1, 2, 20, 30);

console.log(result); // 1번
console.log(arr); // 2번
```

▼ 답

1번 : [2,3] //제거한 요소는 배열로 반환된다.

2번 : [1,20.30,4] //인덱스 1부터 2개 제거후 20,30 넣기

```
const arr = [1, 2, 3, 1, 2];

function remove(array, item) {
  const index = array.indexOf(item);

  if (index !== -1) array.splice(index, 1);

  return array;
}

console.log(remove(arr, 2)); // 1번
console.log(remove(arr, 10)); // 2번
```

▼ 답

1번 : [1,3,1,2] / index = 1 왜냐하면 찾고자하는 게 2이이고 2는 인덱스 1과 4에 있는데 그중 첫번째 인덱스를 반환하기 때문이다. 그리고 index는 -1 이 아니기때문에 array.splice(index, 1); 이것을 실행하여 인덱스 1부터 1개 삭제한다. 따라서 결과값 [1,3,1,2]

2번 : [1,3,1,2] / 10이란 값이 없으니 -1반환. index가 -1이니 array반환하여 [1,3,1,2]

▼ Array.prototype.slice

- 인수로 전달된 범위의 요소들을 복사하여 배열로 반환
- 원본 배열 변경되지 않는다.
- 형식
  - start : 복사를 시작할 인덱스. 음수인 경우 배열의 끝에서의 인덱스를 나타냄. 예를들어 slice(-2)는 배열의 마지막 두 개의 요소를 복사하여 배열로 반환
  - end : 복사를 종료할 인덱스. 이 인덱스에 해당하는 요소는 복사되지 않는다. end는 생략가능하며 생략 시 기본 값은 length 프로퍼티(배열의 길이) 값

```
arr.slice(start,end);
```

▼ 문제

```
const arr = [1, 2, 3];

const result=arr.slice(0, 1);
console.log(result); //1번
console.log(arr);//2번
```

▼ 답

1번 : [1] / 인덱스 0부터 1번 이전까지니 1만 복사되어 저장된다.

2번 : [1,2,3] / 원본배열은 변하지 않으므로 그대로 출력

```
const arr = [1, 2, 3];

const result=arr.slice(-1);
console.log(result) //1번
```

```
const result1=arr.slice(-2);
console.log(reuslt1); //2번

console.log(arr); //3번
```

▼ 답

1번 : [3] / 배열의 끝에서부터 요소를 복사하여 바환

2번 : [2,3]

3번 : [1,2,3]

▼ Array.prototype.join

- 원본 배열의 모든 요소를 문자열로 변환한 후, 인수로 전달받은 문자열, 즉 구분자로 연결한 **문자열을 반환**한다.
- 구분자는 생략가능하며 기본 구분자는 콤마이다.

```
const arr = [1, 2, 3, 4];
- 원ㅂ
// 기본 구분자는 ','이다.
// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 기본 구분자 ','
arr.join(); // -> '1,2,3,4';

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 빈문자열로 연결
arr.join(''); // -> '1234'

// 원본 배열 arr의 모든 요소를 문자열로 변환한 후, 구분자 ':'로 연
arr.join(':'); // -> '1:2:3:4'
```

▼ Array.prototype.reverse

- 원본 배열의 순서를 반대로 뒤집는다.
- 원본 배열이 변경

```
const arr = [1, 2, 3];
const result = arr.reverse();

// reverse 메서드는 원본 배열을 직접 변경한다.
```

```
console.log(arr); // [3, 2, 1]
// 반환값은 변경된 배열이다.
console.log(result); // [3, 2, 1]
```

▼ Array.prototype.**fill**

- 인수로 전달받은 값을 배열의 처음부터 끝까지 요소로 채운다.
- 원본 배열 변경
- 형식

```
arr.fill(배열에 채울 값, 요소 채우기를 시작할 인덱스, 요소 채우
```

```
const arr = [1, 2, 3];

// 인수로 전달 받은 값 0을 배열의 처음부터 끝까지 요소로 채운다.
arr.fill(0);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [0, 0, 0]
```

```
const arr = [1, 2, 3];

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 끝까지 요소로 채운
arr.fill(0, 1);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 0, 0]
```

```
const arr = [1, 2, 3, 4, 5];

// 인수로 전달받은 값 0을 배열의 인덱스 1부터 3 이전(인덱스 3 
arr.fill(0, 1, 3);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // [1, 0, 0, 4, 5]
```

- 단점 : 하나의 값으로만 채울 수 밖에 없다.

▼ 문제

```
const arr = new Array(3);
console.log(arr); // [empty × 3]

// 인수로 전달받은 값 1을 배열의 처음부터 끝까지 요소로 채운다.
const result = arr.fill(1);

// fill 메서드는 원본 배열을 직접 변경한다.
console.log(arr); // 1번

// fill 메서드는 변경된 원본 배열을 반환한다.
console.log(result); //2번
```

▼ 답

1번 : [1,1,1]

2번 : [1,1,1]

▼ Array.prototype.includes

- ES7에 도입
- 배열 내에 특정 요소가 포함되어 있는지 확인하여 true 또는 false반환
- 첫번째 인수로 검색할 대사을 지정

```
const arr = [1, 2, 3];

// 배열에 요소 2가 포함되어 있는지 확인한다.
arr.includes(2); // -> true

// 배열에 요소 100이 포함되어 있는지 확인한다.
arr.includes(100); // -> false
```

```
const arr = [1, 2, 3];

// 배열에 요소 1이 포함되어 있는지 인덱스 1부터 확인한다.
arr.includes(1, 1); // -> false
```

```
// 배열에 요소 3이 포함되어 있는지 인덱스 2(arr.length - 1)부터
arr.includes(3, -1); // -> true
```

▼ Array.prototype.flat

- ES10에서 도입되었다.
- 인수로 전달한 깊이만큼 재귀적으로 배열을 평탄화한다.
- 중첩 배열을 평탄화할 깊이를 인수로 전달할 수 있다. 인수를 생략할 경우 기본값은 1
- 인수로 Infinity를 전달하면 중첩 배열 모두를 평탄화한다.

```
// 중첩 배열을 평탄화하기 위한 깊이 값의 기본값은 1이다.
[1, [2, [3, [4]]]].flat();  // -> [1, 2, [3, [4]]]
[1, [2, [3, [4]]]].flat(1); // -> [1, 2, [3, [4]]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 2로 지정하여 2단계 깊이까지
[1, [2, [3, [4]]]].flat(2); // -> [1, 2, 3, [4]]
// 2번 평탄화한 것과 동일하다.
[1, [2, [3, [4]]]].flat().flat(); // -> [1, 2, 3, [4]]

// 중첩 배열을 평탄화하기 위한 깊이 값을 Infinity로 지정하여 중첩
[1, [2, [3, [4]]]].flat(Infinity); // -> [1, 2, 3, 4]
```

▼ 배열 고차함수

- 고차함수 : 함수를 인수로 전달받거나 함수를 반환하는 함수를 뜻한
- 외부 상태의 변경이나 가변 데이터를 피하고 불변성을 지향하는 함수형 프로그래밍에 기반을 두고 있다.
- **부수효과를 최대한 억제하여 오류를 피하고 프로그램의 안정을 높이려는 노력의 일환**

▼ Array.prototype.sort

- 배열의 요소를 정렬
- 원본 배열을 직접 변경
- 기본값 : 오름차순
- 숫자타입을 정렬할 때 주의 필요

- 배열의 요소가 숫자 타입이라도 문자열로 변환한 후 유니코드 포인트의 순서를 기준으로 정렬한다.

```
const points = [40, 100, 1, 5, 2, 25, 10];
poinsts.sort();
console.log(points);//[1,10,100,2,25,40,5]
```

- 따라서 **정렬 순서를 정의하는 비교 함수를 인수**로 전달해야한다.

```
const points = [40, 100, 1, 5, 2, 25, 10];

// 숫자 배열의 오름차순 정렬. 비교 함수의 반환값이 0보다 작으
points.sort((a, b) => a - b);
console.log(points); // [1, 2, 5, 10, 25, 40, 100]
```

▼ Array.prototype.forEach

- for문을 대체할 수 있는 고차 함수
- 내부에서 반복문을 통해 자신을 호출한 배열을 순회하면서 수행해야 할 처리를 **콜백함수로 전달받아 반복 호출**한다.

```
const numbers = [1, 2, 3];
let pows = [];

// forEach 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함
numbers.forEach(item => pows.push(item ** 2));
console.log(pows); // [1, 4, 9]
```

```
// forEach 메서드는 콜백 함수를 호출하면서 3개(요소값, 인덱스, th
[1, 2, 3].forEach((item, index, arr) => {
  console.log(`요소값: ${item}, 인덱스: ${index}, this: ${
});
/*
요소값: 1, 인덱스: 0, this: [1,2,3]
요소값: 2, 인덱스: 1, this: [1,2,3]
요소값: 3, 인덱스: 2, this: [1,2,3]
*/
```

▼ Array.prototype.map

- 자시을 호출한 배열의 모든 요소를 순회하면서 인수로 전달받은 콜백함수를 반복 호출한다.
- 콜백함수의 반환값들로 구성된 새로운 배열을 반환한다.
- 원본 배열 변경되지 않음

▼ forEach와 차이점

forEach는 언제나 undefiend를 반환하고 map 메서드는 콜백함수의 반환값들로 구성된 새로운 배열을 반환

```
const numbers = [1, 4, 9];

// map 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함수를
// 그리고 콜백 함수의 반환값들로 구성된 새로운 배열을 반환한다.
const roots = numbers.map(item => Math.sqrt(item));

// 위 코드는 다음과 같다.
// const roots = numbers.map(Math.sqrt);

// map 메서드는 새로운 배열을 반환한다
console.log(roots);   // [ 1, 2, 3 ]
// map 메서드는 원본 배열을 변경하지 않는다
console.log(numbers); // [ 1, 4, 9 ]
```

▼ Array.prototype.filter

자신을 호출한 배열을 순회하면서 콜백 함수의 반환값이 true인 요소로만 구성된 새로운 배열을 반환한다.

```
const numbers = [1, 2, 3, 4, 5];

// filter 메서드는 numbers 배열의 모든 요소를 순회하면서 콜백 함
// 그리고 콜백 함수의 반환값이 true인 요소로만 구성된 새로운 배열을
// 다음의 경우 numbers 배열에서 홀수인 요소만을 필터링한다(1은 tru
//item을 2로 나눈 나머지가 0이면 짝수 1이면 홀수인데여기서 0은 fa
const odds = numbers.filter(item => item % 2);
console.log(odds); // [1, 3, 5]
```

▼ Array.prototype.reduce

- 자신을 호출한 배열을 순회하며 인수로 전달받은 콜백 함수를 반복 호출한다. 그리고 콜백 함수의 반환값을 다음 순회 시에 콜백 함수의 첫번 째 인수로 전달하여 하나의 겨과값을 만들어 반환한다.
- 원본 배열 변경 X

```
// [1, 2, 3, 4]의 모든 요소의 누적을 구한다.
const sum = [1, 2, 3, 4].reduce((accumulator, currentVal

console.log(sum); // 10
```

구분	콜백 함수에 전달되는 인수				콜백 함수의 반환값
	accumulator	currentValue	index	array	
첫 번째 순회	0 (초기값)	1	0	[1, 2, 3, 4]	1 (accumulator + currentValue)
두 번째 순회	1	2	1	[1, 2, 3, 4]	3 (accumulator + currentValue)
세 번째 순회	3	3	2	[1, 2, 3, 4]	6 (accumulator + currentValue)
네 번째 순회	6	4	3	[1, 2, 3, 4]	10 (accumulator + currentValue)

▼ Array.prototype.some

- 배열의 요소를 수노히마녀서 인수로 전달된 콜백함수를 호출한다.
- 반환값이 단 한 번이라도 참이면 true, 모두 거짓이면 false를 반환한다.
- 빈배열인 경우 언제라 false반환

```
// 배열의 요소 중에 10보다 큰 요소가 1개 이상 존재하는지 확인
[5, 10, 15].some(item => item > 10); // -> true

// 배열의 요소 중에 0보다 작은 요소가 1개 이상 존재하는지 확인
[5, 10, 15].some(item => item < 0); // -> false
```

```
// 배열의 요소 중에 'banana'가 1개 이상 존재하는지 확인
['apple', 'banana', 'mango'].some(item => item === 'bana

// some 메서드를 호출한 배열이 빈 배열인 경우 언제나 false를 반환
[].some(item => item > 3); // -> false
```

▼ Array.prototype.every

- 배열의 요소를 순회하면서 인수로 전달된 콜백함수를 호출
- 반환값이 모두 참이면 true, 한번이라도 거짓이면 false
- 빈 배열인 경우 true 반환

```
// 배열의 모든 요소가 3보다 큰지 확인
[5, 10, 15].every(item => item > 3); // -> true

// 배열의 모든 요소가 10보다 큰지 확인
[5, 10, 15].every(item => item > 10); // -> false

// every 메서드를 호출한 배열이 빈 배열인 경우 언제나 true를 반환
[].every(item => item > 3); // -> true
```

▼ Array.prototype.find

- 배열의 요소를 순회하면서 인수로 전달된 콜백함수를 호출하여 반환값이 true인 첫 번째 요소를 반환

```
const users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// id가 2인 첫 번째 요소를 반환한다. find 메서드는 배열이 아니라
users.find(user => user.id === 2); // -> {id: 2, name: '

//filter는 배열을 반환한다.
```

```
[1, 2, 2, 3].filter(item => item === 2); // -> [2, 2]

//find는 요소를 반환한다.
[1, 2, 2, 3].find(item => item === 2); // -> 2
```

▼ Array.prototype.findIndex

- 배열의 요소를 순회하면서 인수로 전달된 콜백함수를 호출하여 **반환값이 true인 첫 번째 요소의 *인덱스***를 반환

```
onst users = [
  { id: 1, name: 'Lee' },
  { id: 2, name: 'Kim' },
  { id: 2, name: 'Choi' },
  { id: 3, name: 'Park' }
];

// id가 2인 요소의 인덱스를 구한다.
users.findIndex(user => user.id === 2); // -> 1

// name이 'Park'인 요소의 인덱스를 구한다.
users.findIndex(user => user.name === 'Park'); // -> 3

// 위와 같이 프로퍼티 키와 프로퍼티 값으로 요소의 인덱스를 구하는 경
// 다음과 같이 콜백 함수를 추상화할 수 있다.
function predicate(key, value) {
  // key와 value를 기억하는 클로저를 반환
  return item => item[key] === value;
}

// id가 2인 요소의 인덱스를 구한다.
users.findIndex(predicate('id', 2)); // -> 1

// name이 'Park'인 요소의 인덱스를 구한다.
users.findIndex(predicate('name', 'Park')); // -> 3
```

▼ Array.prototype.flatMap

map 메서드를 통해 생성된 새로운 배열을 평탄화한다.

즉, map 메서드와 flat 메서드를 순차적으로 실행한다.

```
const arr = ['hello', 'world'];

// map과 flat을 순차적으로 실행
arr.map(x => x.split('')).flat();
// -> ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']

// flatMap은 map을 통해 생성된 새로운 배열을 평탄화한다.
arr.flatMap(x => x.split(''));
// -> ['h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd']
const arr = ['hello', 'world'];

// flatMap은 1단계만 평탄화한다.
arr.flatMap((str, index) => [index, [str, str.length]]);
// -> [[0, ['hello', 5]], [1, ['world', 5]]] => [0, ['he

// 평탄화 깊이를 지정해야 하면 flatMap 메서드를 사용하지 말고 map
arr.map((str, index) => [index, [str, str.length]]).flat
// -> [[0, ['hello', 5]], [1, ['world', 5]]] => [0, 'hel
```