09. 배열 내장함수

이번에는 배열을 다룰 때 알고있으면 너무나 유용한 다양한 내장 함수들에 대하여 알아보겠습니다.

forEach

`forEach` 는 가장 쉬운 배열 내장함수입니다. 기존에 우리가 배웠던 for 문을 대체 시킬 수 있습니다. 예를 들어서 다음과 같은 텍스트 배열이 있다고 가정해봅시다.

```
const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];
```

만약, 배열 안에 있는 모든 원소들을 모두 출력해야 한다면 for 문을 사용하여 다음과 같이 구현 할 수 있는데요,

```
const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];

for (let i = 0; i < superheroes.length; i++) {
  console.log(superheroes[i]);
}
```

배열의 forEach 함수를 사용하면 다음과 같이 구현 할 수 있습니다.

```
const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];

superheroes.forEach(hero => {
  console.log(hero);
});
```

forEach 함수의 파라미터로는, 각 원소에 대하여 처리하고 싶은 코드를 함수로 넣어줍니다. 이 함수의 파라미터 hero는 각 원소를 가르키게 됩니다.

이렇게 함수형태의 파라미터를 전달하는 것을 콜백함수 라고 부릅니다. 함수를 등록해주면, forEach 가 실행을 해주는 거죠.

map

`map` 은 배열 안의 각 원소를 변환 할 때 사용 되며, 이 과정에서 새로운 배열이 만들어집니다.

예를 들어서 다음과 같은 배열이 있다고 가정해봅시다.

```
const array = [1, 2, 3, 4, 5, 6, 7, 8];
```

만약에 배열 안의 모든 숫자를 제곱해서 새로운 배열을 만들고 싶다면 어떻게 해야 할까요? map 함수를 사용하지 않고 우리가 지금까지 배운 지식들을 활용하면 다음과 같이 구현 할 수 있습니다.

```
const array = [1, 2, 3, 4, 5, 6, 7, 8];

const squared = [];
for (let i = 0; i < array.length; i++) {
  squared.push(array[i] * array[i]);
}

console.log(squared);
```

또는 방금 배운 forEach 를 쓰면 다음과 같이 구현 할 수도 있겠죠

```
const array = [1, 2, 3, 4, 5, 6, 7, 8];

const squared = [];

array.forEach(n => {
  squared.push(n * n);
});

console.log(squared);
```

결과는 다음과 같습니다.

```
[1, 4, 9, 16, 25, 36, 49, 64];
```

만약 map 을 사용하면 이를 더 짧은 코드를 사용하여 구현 할 수 있습니다.

```
const array = [1, 2, 3, 4, 5, 6, 7, 8];

const square = n => n * n;
const squared = array.map(square);
console.log(squared);
```

똑같은 결과가 나타났나요?

map 함수의 파라미터로는 변화를 주는 함수를 전달해줍니다. 이를 변화함수라고 부르도록 하겠습니다.

현재 우리의 변화함수 square 는 파라미터 n 을 받아와서 이를 제곱해줍니다.

array.map 함수를 사용 할 때 square 를 변화함수로 사용함으로서, 내부의 모든 값에 대하여 제곱을 해서 새로운 배열을 생성하였습니다.

변화 함수를 꼭 이름을 붙여서 선언 할 필요는 없습니다. 코드를 다음과 같이 작성해도 됩니다.

```
const squared = array.map(n => n * n);
console.log(squared);
```

indexOf

`indexOf` 는 원하는 항목이 몇번째 원소인지 찾아주는 함수입니다.

예를 들어서 다음과 같은 배열이 있을 때

```
const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];
```

토르가 몇번째 항목인지 알고싶다고 가정해봅시다.

그렇다면, 이렇게 입력 할 수 있습니다.

```
const superheroes = ['아이언맨', '캡틴 아메리카', '토르', '닥터 스트레인지'];
const index = superheroes.indexOf('토르');
console.log(index);
```

결과는 2가 나타납니다.

index 값은 0 부터 시작하기 때문에 0: 아이언맨 1: 캡틴 아메리카 2: 토르

이렇게 돼서 2라는 값이 나타나는 것 입니다.

findIndex

만약에 배열 안에 있는 값이 숫자, 문자열, 또는 불리언이라면 찾고자하는 항목이 몇번째 원소인지 알아내려면 indexOf 를 사용하면 됩니다. 하지만, 배열 안에 있는 값이 객체이거나, 배열이라면 indexOf 로 찾을 수 없습니다.

예를 들어서 다음과 같은 배열이 있다고 가정해봅시다.

```
const todos = [
  {
    id: 1,
    text: '자바스크립트 입문',
    done: true
  },
  {
    id: 2,
    text: '함수 배우기',
    done: true
  },
  {
    id: 3,
    text: '객체와 배열 배우기',
    done: true
  },
  {
    id: 4,
    text: '배열 내장함수 배우기',
    done: false
  }
];
```

여기서 만약 id 가 3 인 객체가 몇번째인지 찾으려면, `findIndex` 함수에 검사하고자 하는 조건을 반환하는 함수를 넣어서 찾을 수 있습니다.

```
const todos = [
  {
    id: 1,
    text: '자바스크립트 입문',
    done: true
  },
  {
    id: 2,
    text: '함수 배우기',
    done: true
  },
  {
    id: 3,
    text: '객체와 배열 배우기',
    done: true
  },
  {
    id: 4,
    text: '배열 내장함수 배우기',
    done: false
  }
];

const index = todos.findIndex(todo => todo.id === 3);
console.log(index);
```

결과는 2가 나타납니다.

find

`find` 함수는 `findIndex` 랑 비슷한데, 찾아낸 값이 몇번째인지 알아내는 것이 아니라, 찾아낸 값 자체를 반환합니다.

```
const todos = [
  {
    id: 1,
    text: '자바스크립트 입문',
    done: true
  },
  {
    id: 2,
    text: '함수 배우기',
    done: true
  },
  {
    id: 3,
    text: '객체와 배열 배우기',
    done: true
  },
  {
    id: 4,
    text: '배열 내장함수 배우기',
    done: false
  }
];

const todo = todos.find(todo => todo.id === 3);
console.log(todo);
```

결과는 다음과 같습니다.

```
{id: 3, text: "객체와 배열 배우기", done: true}
```

filter

`filter` 함수는 배열에서 특정 조건을 만족하는 값들만 따로 추출하여 새로운 배열을 만듭니다. 예를 들어서, 우리가 방금 만들었던 todos 배열에서 done 값이 false 인 항목들만 따로 추출해서 새로운 배열을 만들어봅시다.

```
const todos = [
  {
    id: 1,
    text: '자바스크립트 입문',
    done: true
  },
  {
    id: 2,
    text: '함수 배우기',
    done: true
  },
  {
    id: 3,
    text: '객체와 배열 배우기',
    done: true
  },
  {
    id: 4,
    text: '배열 내장함수 배우기',
    done: false
  }
];

const tasksNotDone = todos.filter(todo => todo.done === false);
console.log(tasksNotDone);
```

결과는 다음과 같습니다.

```
[
  {
    id: 4,
    text: '배열 내장 함수 배우기',
    done: false
  }
];
```

filter 함수에 넣는 파라미터는 조건을 검사하는 함수를 넣어주며, 이 함수의 파라미터로 각 원소의 값을 받아오게 됩니다.

방금 우리가 작성한 코드는 이렇게 입력 할 수도 있습니다.

```
const tasksNotDone = todos.filter(todo => !todo.done);
```

filter 에 넣어준 함수에서 true 를 반환하면 새로운 배열에 따로 추출을 해주는데요, 만약 todo.done 값이 false 라면, !false 가 되고 이 값은 true 이기 때문에, 이전의 todo.done === false 와 똑같이 작동하게 됩니다.

splice

splice 는 배열에서 특정 항목을 제거할 때 사용합니다.

```
const numbers = [10, 20, 30, 40];
```

위 배열에서 30 을 지운다고 가정해봅시다. 그러면, 30이 몇번째 index 인지 알아낸 이후, 이를 splice 를 통해 지워줄 수 있습니다.

```
const numbers = [10, 20, 30, 40];
const index = numbers.indexOf(30);
numbers.splice(index, 1);
console.log(numbers);
```

결과는 다음과 같습니다.

```
[10, 20, 40]
```

splice 를 사용 할 때 첫번째 파라미터는 어떤 인덱스부터 지울지를 의미하고 두번째 파라미터는 그 인덱스부터 몇개를 지울지를 의미합니다.

slice

slice 는 splice 랑 조금 비슷한데요, 배열을 잘라낼 때 사용하는데, 중요한 점은 기존의 배열은 건들이지 않는 다는 것입니다.

```
const numbers = [10, 20, 30, 40];
const sliced = numbers.slice(0, 2); // 0부터 시작해서 2전까지

console.log(sliced); // [10, 20]
console.log(numbers); // [10, 20, 30, 40]
```

slice 에는 두개의 파라미터를 넣게 되는데 첫번째 파라미터는 어디서부터 자를지, 그리고 두번째 파라미터는 어디까지 자를지 를 의미합니다.

shift 와 pop

`shift` 와 `pop` 은 비슷하지만, 다릅니다.

`shift` 는 첫번째 원소를 배열에서 추출해줍니다. (추출하는 과정에서 배열에서 해당 원소는 사라집니다.)

```
const numbers = [10, 20, 30, 40];
const value = numbers.shift();
console.log(value);
console.log(numbers);
```

결과는 다음과 같습니다.

```
10
[20, 30, 40]
```

이번엔 pop 을 해볼까요?

```
const numbers = [10, 20, 30, 40];
const value = numbers.pop();
console.log(value);
console.log(numbers);
```

결과는 다음과 같습니다.

```
40
[10, 20, 30]
```

pop 은 push 의 반대로 생각하시면 됩니다. push 는 배열의 맨 마지막에 새 항목을 추가하고, pop 은 맨 마지막 항목을 추출합니다.

unshift

unshift 는 shift 의 반대입니다.

배열의 맨 앞에 새 원소를 추가합니다.

```
const numbers = [10, 20, 30, 40];
numbers.unshift(5);
console.log(numbers);
```

결과는 다음과 같습니다.

```
[5, 10, 20, 30, 40]
```

concat

concat 은 여러개의 배열을 하나의 배열로 합쳐줍니다.

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const concated = arr1.concat(arr2);

console.log(concated);
```

결과는 다음과 같습니다.

```
[1, 2, 3, 4, 5, 6];
```

concat 함수는 arr1 과 arr2 에 변화를 주지 않습니다.

join

join 은 배열 안의 값들을 문자열 형태로 합쳐줍니다.

```
const array = [1, 2, 3, 4, 5];
console.log(array.join()); // 1,2,3,4,5
console.log(array.join(' ')); // 1 2 3 4 5
console.log(array.join(', ')); // 1, 2, 3, 4, 5
```

reduce

reduce 함수는 잘 사용 할 줄 알면 정말 유용한 내장 함수입니다. 만약 여러분이 주어진 배열에 대하여 총합을 구해야 하는 상황이 왔다고 가정해봅시다.

이렇게 구현을 할 수 있을텐데요

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

let sum = 0;
numbers.forEach(n => {
  sum += n;
});
console.log(sum);
```

(결과는 15가 됩니다)

여기서 sum 을 계산하기 위해서 사전에 sum 을 선언하고, forEach 를 통하여 계속해서 덧셈을 해주었는데요,

`reduce` 라는 함수를 사용하면 다음과 같이 구현 할 수 있습니다.

```
const numbers = [1, 2, 3, 4, 5];
let sum = array.reduce((accumulator, current) => accumulator + current, 0);

console.log(sum);
```

reduce 함수에는 두개의 파라미터를 전달합니다. 첫번째 파라미터는 accumulator 와 current 를 파라미터로 가져와서 결과를 반환하는 콜백함수이구요, 두번째 파라미터는 reduce 함수에서 사용 할 초깃값입니다.

여기서 accumulator 는 누적된 값을 의미합니다.

방금 작성한 함수를 다음과 같이 수정해보세요.

```
const numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((accumulator, current) => {
  console.log({ accumulator, current });
  return accumulator + current;
}, 0);

console.log(sum);
```

이 코드의 실행 결과는 다음과 같습니다.

```
▶Object {accumulator: 0, current: 1}
▶Object {accumulator: 1, current: 2}
▶Object {accumulator: 3, current: 3}
▶Object {accumulator: 6, current: 4}
▶Object {accumulator: 10, current: 5}
15
```

배열을 처음부터 끝까지 반복하면서 우리가 전달한 콜백 함수가 호출이 되는데요, 가장 처음엔 accumulator 값이 0 입니다. 이 값이 0 인 이유는 우리가 두번째 파라미터인 초깃값으로 0을 설정했기 때문입니다.

처음 콜백 함수가 호출되면, 0 + 1 을 해서 1이 반환됩니다. 이렇게 1일 반환되면 그 다음 번에 콜백함수가 호출 될 때 accumulator 값으로 사용됩니다.

콜백함수가 두번째로 호출 될 땐 1 + 2 를 해서 3이되고, 이 값이 세번째로 호출될 때의 accumulator 가 됩니다.

그래서 쭉- 누적돼서 결과물 15가 나타나는 것 입니다.

reduce 를 사용해서 평균도 계산 할 수 있습니다. 평균을 계산하려면, 가장 마지막 숫자를 더하고 나서 배열의 length 로 나누어주어야 합니다.

```
const numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce((accumulator, current, index, array) => {
  if (index === array.length - 1) {
    return (accumulator + current) / array.length;
  }
  return accumulator + current;
}, 0);

console.log(sum);
```

결과는 3이 됩니다.

위 코드의 reduce 에서 사용한 콜백함수에서는 추가 파라미터로 index 와 array 를 받아왔습니다. index 는 현재 처리하고 있는 항목이 몇번째인지 가르키고, array 는 현재 처리하고 있는 배열 자신을 의미합니다.

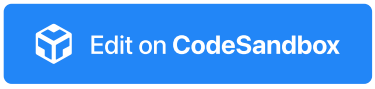
퀴즈

이제 지금까지 배운 것들을 활용하여 퀴즈를 풀어봅시다!

숫자 배열이 주어졌을 때 10보다 큰 숫자의 갯수를 반환하는 함수를 만드세요.

```
function countBiggerThanTen(numbers) {
  /* 구현해보세요 */
}

const count = countBiggerThanTen([1, 2, 3, 5, 10, 20, 30, 40, 50, 60]);
console.log(count); // 5
```



위 버튼을 클릭하여 코드를 작성해서 Test를 통과시키세요.

이 문제는 우리가 배운 내장 함수들 중에서 여러 종류를 사용하여 다른 방식으로 구현 할 수 있습니다