230531_실습

JavaScript

모듈

모듈이란?

- 프로그램을 구성하는 구성 요소로, 관련된 데이터와 함수를 하나로 묶은 단위를 의미
    - Framework Package Library Module

import : 모듈 불러오기

```
import "module-name";
```

- “module-name” 경로의 파일에서 모듈을 불러온다.
- 확장자(.js)는 생략 가능하다.

```
import {myMember} from "my-module.js";
```

```
import {foo, bar} from "my-module.js";
```

- "my-module.js" 파일에서 해당하는 부분(myMember, foo, bar)을 가져와 각각의 이름의 변수에 할당한다. → 구조분해할당

```
import * as name from "module-name";
```

- “module-name”에서 모듈 전체(*)를 가져와서 “name” 변수에 할당한다.
  - “module-name”에서 모듈 전체(*)를 가져와서 “name” 변수에 할당한다.```jsx
  import * as name from "module-name";
  ```

  - “module-name”에서 모듈 전체(*)를 가져와서 “name” 변수에 할당한다.
  ```

```
import {sayHi as hi, sayBye as bye} from './say.js';
import * as say from './say.js';
```

export : 모듈 내보내기

```
// 하나씩 내보내기
    export let name1, name2, …, nameN; // var, const도 동일
    export let name1 = …, name2 = …, …, nameN; // var, const도 동일
    export function functionName(){...}
    export class ClassName {...}

    // 목록으로 내보내기
    export { name1, name2, …, nameN };

    // 내보내면서 이름 바꾸기
    export { variable1 as name1, variable2 as name2, …, nameN };

    // 비구조화로 내보내기
    export const { name1, name2: bar } = o;

    // 기본 내보내기
    export default expression;
    export default function (…) { … } // also class, function*
    export default function name1(…) { … } // also class, function*
    export { name1 as default, … };

    // 모듈 조합
    export * from …; // does not set the default export
    export * as name1 from …;
    export { name1, name2, …, nameN } from …;
    export { import1 as name1, import2 as name2, …, nameN } from …;
    export { default } from …;
```

Expression

template literal

- 템플릿 리터럴은 내장된 표현식을 허용하는 문자열 리터럴입니다.
- **표현식/문자열 삽입, 여러 줄 문자열, 문자열 형식화, 문자열 태깅** 등 다양한 기능을 제공합니다.
- 일반 문자열

```
var a = "자바스크립트";
var result = a + " 누가 만들었어?——";
console.log(result);
//자바스크립트 누가 만들었어?——
```

- 템플릿 리터럴

```
let a = "자바스크립트";
let result = `${a} 누가 만들었어?ㅡㅡ`;
console.log(result);
```

arrow function expression

- 함수 표현식보다 단순하고 간결한 문법으로 함수를 만들 수 있는 방법
- 기본 함수

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

- 화살표 함수

```
let func = (arg1, arg2, ...argN) => expression
```

```
let func = arg1 => expression;

let double = n => n * 2;
```

```
let func= () => expression;
```

- Quiz
  - 다음 함수를 화살표 함수로 변환하세요

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "동의하십니까?",
  function() { alert("동의하셨습니다."); },
  function() { alert("취소 버튼을 누르셨습니다."); }
);
```

- 정답

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask(
  "동의하십니까?",
  () => alert("동의하셨습니다."),
  () => alert("취소 버튼을 누르셨습니다.")
);
```

this

this 란?

- this란 ‘이것’ 이란 뜻이다.
- this란 JavaScript 예약어다.

this 는?

- this는 자신이 속한 객체 또는 자신이 생성할 인스턴스를 가리키는 자기 참조 변수(self-reference variable)이다.
- this를 통해 자신이 속한 객체 또는 자신이 생성할 인스턴스의 프로퍼티나 메서드를 참조할 수 있다.
- this는 자바스크립트 엔진에 의해 암묵적으로 생성된다.
- this는 코드 어디서든 참조할 수 있다.
- 하지만 this는 객체의 프로퍼티나 메서드를 참조하기 위한 자기 참조 변수이므로 일반적으로 객체의 메서드 내부 또는 생성자 함수 내부에서만 의미가 있다.
- 함수를 호출하면 인자와 this가 암묵적으로 함수 내부에 전달된다.
- 함수 내부에서 인자를 지역 변수처럼 사용할 수 있는 것처럼, this도 지역 변수처럼 사용할 수 있다.
- 단, this가 가리키는 값, 즉 this 바인딩은 함수 호출 방식에 의해 동적으로 결정된다.
- 크게 전역에서 사용할 때와 함수안에서 사용할 때로 나눌 수 있다.
- this는 이처럼 어떤 위치에 있느냐, 어디에서 호출하느냐, 어떤 함수에 있느냐에 따라 참조 값이 달라지는 특성이 있다. 그래서 사용할 때 주의해야한다.

binding이란?

- 식별자와 값을 연결하는 과정을 말한다.
- 변수선언은 변수 이름과 확보된 메모리 공간의 주소를 바인딩하는 것이다.
- this 바인딩은 this(키워드로 분류되지만 식별자의 역할을 한다.)와 this가 가리킬 객체를 바인딩하는 것이다.
- 예제 1
  - 바인딩이 되지 않은 경우

```
let A = {
    prop: 'Hello',
    sayHello: function() {
        console.log( this.prop );
    }
};

let B = A.sayHello();
console.log(B); // undefined
```

  - 바인딩을 사용한 경우

```
let A = {
    prop: 'Hello',
    sayHello: function() {
        console.log( this.prop );
    }
};

let B = A.sayHello.bind(A);
console.log(B); // function() {
        //console.log( this.prop );
    //}
```

- 예제 2

```
const foo = {
  a: 20,
  bar: function () {
    setTimeout(function () {
      console.log(this.a);
    }, 1);
  }
}

foo.bar(); // undefined
```

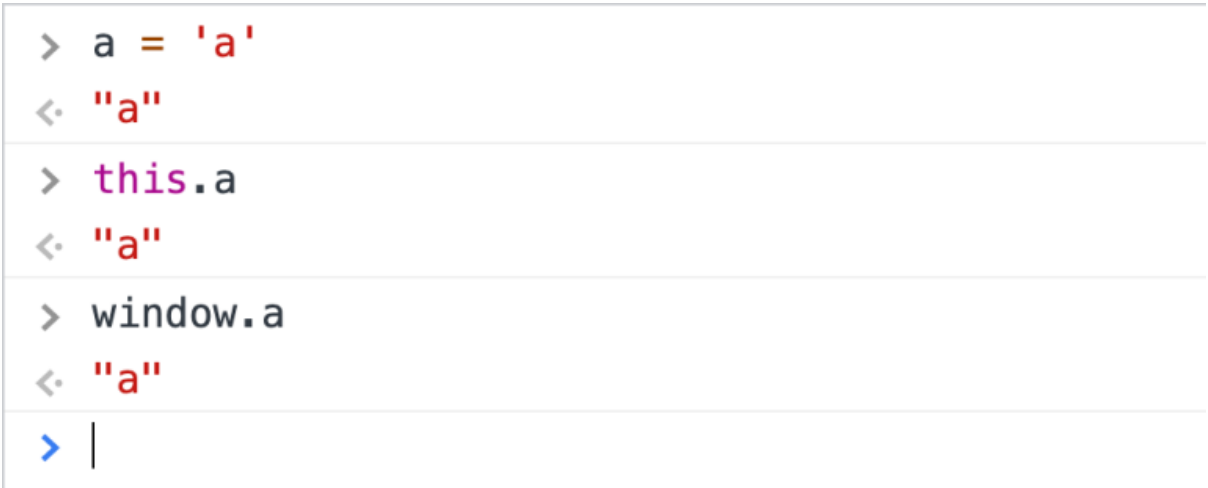
```
const foo = {
  a: 20,
  bar: function () {
    const _this = this;
    setTimeout(function () {
      console.log(_this.a);
    }, 1);
  }
}

foo.bar(); // 20
```

this 사용

단독으로 쓴 this(this를 전역에서 사용한 경우)

- 브라우저라는 자바스크립트 런타임의 경우에 this는 항상 window라는 전역 객체를 참조한다.
- 전역 객체란 전역 범위에 항상 존재하는 객체를 의미한다. (Node.js에서 전역 객체는 global 이다.)
- 브라우저라는 자바스크립트 런타임에서 모든 변수, 함수는 window라는 객체의 프로퍼티와 메소드이다.



- 브라우저의 전역 객체 window

```
~ node
Welcome to Node.js v14.16.0.
Type ".help" for more information.
> this
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  }
}
> global
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Function (anonymous)]
  }
}
>
```

- Node.js 의 전역 객체 global
- Node.js의 REPL에 this 와 global 키워드를 각각 입력한 결과가 동일한 것을 확인 할 수 있다.
- 단독으로 this를 호출하는 경우엔 global object를 가리킵니다.
- 브라우저에서 호출하는 경우 [object Window](ES5에서 추가된 strict mode(엄격 모드)에서도 동일)


```
'use strict';

var x = this;
console.log(x); //Window
```

함수 안에서 쓴 this(this를 함수 내부에서 사용한 경우)

- 함수는 전역에 선언된 일반 함수와 객체 안에 메소드로 크게 구분할 수 있다.
- 객체안에 선언된 함수를 전역에 선언된 함수와 구분하기 위해 메소드라고 한다.
- 그런데 전역에 선언된 일반 함수도 결국 window 전역 객체의 메소드다.
- 즉, 모든 함수는 객체 내부에 있다.
- 이때 this는 현재 함수를 실행하고 있는 그 객체를 참조한다.
- 정리하면 함수 내부에서 this의 값은 함수를 호출하는 방법에 의해 바뀐다.
- 그리고 또한 엄격모드 여부에 따라 참조 값이 달라진다.엄격 모드에서 일반 함수 내부의 this는 undefinded 가 바인딩 된다.
- 함수 안에서 this는 함수의 주인에게 바인딩됩니다. 함수의 주인은? window객체죠!

```
function myFunction() {
  return this;
}
console.log(myFunction()); //Window
```

```
var num = 0;
function addNum() {
  this.num = 100;
  num++;

  console.log(num); // 101
  console.log(window.num); // 101
  console.log(num === window.num); // true
}

addNum();
```

전역에 선언된 함수에서 this

1. Case 1.
    - function → global (window, global) 인 경우

```
function myFn () {
  return this;
}
myFn(); // window 객체 출력
```

```
function myFunction() {
  return this;
}
console.log(myFunction()); //Window
```

```
var num = 0;
function addNum() {
  this.num = 100;
  num++;

  console.log(num); // 101
  console.log(window.num); // 101
  console.log(num === window.num); // true
}

addNum();
```

- 위 코드에서 this.num의 this는 window 객체를 가리킵니다.
- 따라서 num은 전역 변수를 가리키게 됩니다.
- 다만, strict mode(엄격 모드)에서는 조금 다릅니다.
- 함수 내의 this에 디폴트 바인딩이 없기 때문에 undefined가 됩니다.

```
"use strict";

function myFunction() {
  return this;
}
console.log(myFunction()); //undefined
```

```
"use strict";

var num = 0;
```

```
function addNum() {
  this.num = 100; //ERROR! Cannot set property 'num' of undefined
  num++;
}

addNum();
```

- 따라서 this.num을 호출하면 undefined.num을 호출하는 것과 마찬가지기 때문에 에러가 납니다.

2. Case 2.
   - new 연산자를 사용해서 생성자 함수방식으로 인스턴스를 생성한 경우이다.
   - 생성자 함수 MyFn가 빈 객체를 만들고 이 생성자 함수에서 this가 이 빈 객체를 가리키도록 설정하였다.

```
function MyFn() {
  this.title = 'Hello World!';
  return this;
}
// new 연산자를 이용해서 새로운 객체를 얻는다.
const myfn = new MyFn();
myfn // MyFn {title: 'Hello World!'}
```

객체의 메소드 함수에서 this

- 메서드 호출 시 메서드 내부 코드에서 사용된 this는 해당 메서드를 호출한 객체로 바인딩됩니다.

```
var person = {
  firstName: 'John',
  lastName: 'Doe',
  fullName: function () {
    return this.firstName + ' ' + this.lastName;
  },
};

person.fullName(); //"John Doe"
```

```
var num = 0;

function showNum() {
  console.log(this.num);
}

showNum(); //0
```

```
var obj = {
  num: 200,
  func: showNum,
};

obj.func(); //200
```

1. Case 1.
   - method → obj 인 경우이다.
   - showTitle() 메소드는 fn 객체의 메소드이기 때문에 this는 fn 객체를 참조한다.

```
const fn = {
  title: 'Hello World!',
  showTitle() {
    console.log(this.title);
  }
};
fn.showTitle(); // 'Hello World!'
```

2. Case 2.
   - 고차 함수의 콜백함수 안에서 this는 콜백함수가 일반 함수이기 때문에 전역 객체를 참조한다.

```
const fn = {
  title: 'Hello World!',
  tags: [1, 2, 3, 4],
  showTags() {
    this.tags.forEach(function(tag) {
      console.log(tag);
      console.log(this); // window
    });
  }
}
fn.showTags();
// 1
// window 객체 출력
// 2
// window 객체 출력
// 3
// window 객체 출력
// 4
// window 객체 출력/n
```

3. Case 3.
   - Case 2. 해결 방법으로 콜백함수 다음 인자로 참조할 객체를 전달해준다.

```
const fn = {
  title: 'Hello World!',
  tags: [1, 2, 3, 4],
  showTags() {
    this.tags.forEach(function(tag) {
      console.log(tag);
      console.log(this); // fn
    }, this); // 여기는 일반 함수 바깥, fn 객체를 참조할 수 있다.
  }
}
fn.showTags();
// 1
// fn 객체 출력
// 2
// fn 객체 출력
// 3
// fn 객체 출력
// 4
// fn 객체 출력
```

4. Case 4.
    - function 키워드로 생성한 일반함수와 화살표 함수의 가장 큰 차이점이 바로 this이다.
    - 이를 Lexical this (렉시컬 this)라고 한다.
    - 화살표 함수 안에서 this는 언제나 상위 스코프의 this를 가리킨다.
    - 일반 함수는 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정되지 않고, 함수를 호출 할 때 함수가 어떻게 호출 되는지에 따라 this에 바인딩할 객체가 동적으로 결정된다.
    - 화살표 함수는 함수를 선언할 때 this에 바인딩할 객체가 정적으로 결정된다.
    - 화살표 함수의 this 바인딩 객체 결정 방식은 함수의 상위 스코프를 결정하는 방식인 렉시컬 스코프와 유사하다.
    - 화살표 함수는 call, apply, bind 메소드를 사용하여 this를 변경할 수 없다.

```
const fn = {
  title: 'Hello World!',
  tags: [1, 2, 3, 4],
  showTags() {
    this.tags.forEach((tag) => {
      console.log(tag);
      console.log(this); // fn
    });
  }
}
fn.showTags();
```

```
// 1
// fn 객체 출력
// 2
// fn 객체 출력
// 3
// fn 객체 출력
// 4
// fn 객체 출력
```

이벤트 핸들러 안에서 쓴 this

- 이벤트 핸들러에서 this는 이벤트를 받는 HTML 요소를 가리킵니다.

```
var btn = document.querySelector('#btn')
btn.addEventListener('click', function () {
  console.log(this); //#btn
});
```

생성자 안에서 쓴 this

- 생성자 함수가 생성하는 객체로 this가 바인딩 됩니다.

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

var kim = new Person('kim');
var lee = new Person('lee');

console.log(kim.name); //kim
console.log(lee.name); //lee
```

- 하지만 new 키워드를 빼먹는 순간 일반 함수 호출과 같아지기 때문에, 이 경우는 this가 window에 바인딩됩니다.

```
var name = 'window';
function Person(name) {
  this.name = name;
}

var kim = Person('kim');

console.log(window.name); //kim
```

명시적 바인딩을 한 this

- 명시적 바인딩은 짝을 지어주는 거에요. 이 this는 내꺼! 같은 거
- apply() 와 call() 메서드는 Function Object에 기본적으로 정의된 메서드인데요, 인자를 this로 만들어주는 기능을 합니다.

```
function whoisThis() {
  console.log(this);
}

whoisThis(); //window

var obj = {
  x: 123,
};

whoisThis.call(obj); //{x:123}
```

- apply()에서 매개변수로 받은 첫 번째 값은 함수 내부에서 사용되는 this에 바인딩되고,
- 두 번째 값인 배열은 자신을 호출한 함수의 인자로 사용합니다.
- 활용

```
function Character(name, level) {
  this.name = name;
  this.level = level;
}

function Player(name, level, job) {
  this.name = name;
  this.level = level;
  this.job = job;
}
```

- 이렇게 두 생성자 함수가 있다고 해봅시다. this.name과 this.level을 받아오는 부분이 똑같습니다.
- 이럴 때 apply()을 쓸 수 있어요.

```
function Character(name, level) {
  this.name = name;
  this.level = level;
}

function Player(name, level, job) {
  Character.apply(this, [name, level]);
  this.job = job;
}
```

```
var me = new Player('Nana', 10, 'Magician');
```

- call()도 apply()와 거의 같습니다.
- 차이점이 있다면 call()은 인수 목록을 받고 apply()는 인수 배열을 받는다는 차이가 있어요.
- 위 코드를 call()로 바꿔 쓴다면 이렇게 되겠죠 :)
- 둘다 일단은 함수를 호출한다는 것에 주의하세요.

```
function Character(name, level) {
  this.name = name;
  this.level = level;
}

function Player(name, level, job) {
  Character.call(this, name, level);
  this.job = job;
}

var me = {};
Player.call(me, 'nana', 10, 'Magician');
```

- apply()나 call()은 보통 유사배열 객체에게 배열 메서드를 쓰고자 할 때 사용합니다.
- 예를 들어 arguments 객체는 함수에 전달된 인수를 Array 형태로 보여주지만 배열 메서드를 쓸 수가 없습니다.
- 이럴 때 쓱하고 가져다 쓸 수 있어요.

```
function func(a, b, c) {
  console.log(arguments);

  arguments.push('hi!'); //ERROR! (arguments.push is not a function);
}
```

```
function func(a, b, c) {
  var args = Array.prototype.slice.apply(arguments);
  args.push('hi!');
  console.dir(args);
}

func(1, 2, 3); // [ 1, 2, 3, 'hi!' ]
```

```
var list = {
  0: 'Kim',
  1: 'Lee',
  2: 'Park',
  length: 3,
};

Array.prototype.push.call(list, 'Choi');
console.log(list);
```

- 추가로 ES6부터 Array.from()이라는 메서드를 쓸 수 있어요.
- 유사배열객체를 얕게 복사해 새 Array 객체로 만듭니다.

```
var children = document.body.children; // HTMLCollection

children.forEach(function (el) {
  el.classList.add('on'); //ERROR! (children.forEach is not a function)
});
```

```
var children = document.body.children; // HTMLCollection

Array.from(children).forEach(function (el) {
  el.classList.add('on');
});
```

화살표 함수로 쓴 this

- 함수 안에서 this가 전역객체가 될 땐 화살표 함수를 쓰면 됩니다.
- 화살표 함수는 전역 컨텍스트에서 실행되더라도 this를 새로 정의하지 않고, 바로 바깥 함수나 클래스의 this를 쓰거든요

```
var Person = function (name, age) {
  this.name = name;
  this.age = age;
  this.say = function () {
    console.log(this); // Person {name: "Nana", age: 28}

    setTimeout(function () {
      console.log(this); // Window
      console.log(this.name + ' is ' + this.age + ' years old');
    }, 100);
  };
};
var me = new Person('Nana', 28);
```

```
me.say(); //global is undefined years old
```

- 위 코드를 보면 내부 함수에서 this가 전역 객체를 가리키는 바람에 의도와는 다른 결과가 나왔습니다.

```
var Person = function (name, age) {
  this.name = name;
  this.age = age;
  this.say = function () {
    console.log(this); // Person {name: "Nana", age: 28}

    setTimeout(() => {
      console.log(this); // Person {name: "Nana", age: 28}
      console.log(this.name + ' is ' + this.age + ' years old');
    }, 100);
  };
};
var me = new Person('Nana', 28); //Nana is 28 years old
```

- 하지만 화살표 함수로 바꾸면 제대로 된 결과가 나오는 걸 볼 수 있습니다.

Closure

closure란?

- 내부 함수에서 외부 함수에 접근 가능한 함수

closure 살펴보기

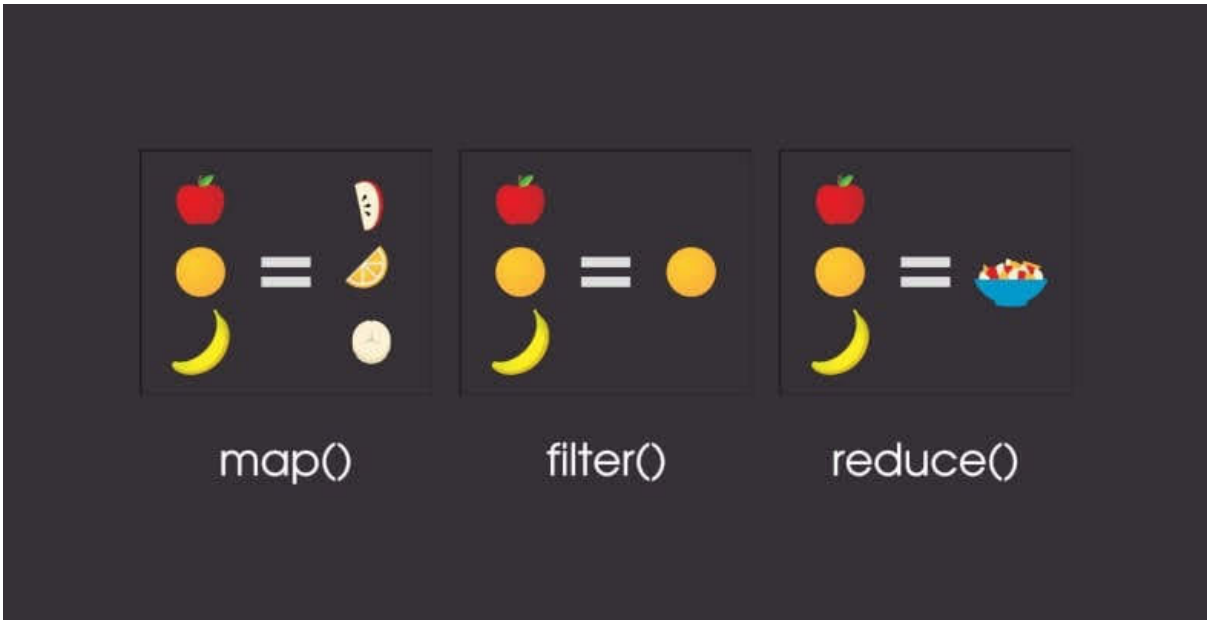
```
function outerFn() {
  let x = 10;

  return function innerFn(y) {
    return x = x + y;
  }
}

let a = outerFn();
a(5); // 15;
a(5); // 20;
a(5); // 25;
```

- 외부 함수 : outerFn()
- 내부 함수 innerFn()

Array Methods



map

- 배열의 각 요소에 대하여 주어진 함수를 수행한 결과를 모아 새로운 배열을 반환하는 메서드
- 예) 단위변환(섭씨→화씨), 숫자→문자 등
- 요소들에게 일괄적으로 함수를 적용하고 싶을 때 사용하기 적합
- 문법

```
const output = arr.map(function);
```

- 예제

```
const numbers = [1, 2, 3, 4, 5];
const numbersMap = numbers.map(val =&gt; val * 2);

console.log(numbersMap);

// [ 2, 4, 6, 8, 10 ]
```

```
const arr = [5, 1, 3, 2, 6];

// double
// [10, 2, 6, 4, 12]

// triple
// [15, 3, 9, 6, 18]

// binary
// ["101, "1", "11", "10", "110"]

function double(x) {
  return x * 2;
}

function triple(x) {
  return x * 3;
}

function binary(x) {
  return x.toString("2");
}

const output = arr.map(double);

const output2 = arr.map(triple);

const output3 = arr.map(binary);

console.log(output);
console.log(output2);
console.log(output3);
```

filter

- 배열의 각 요소에 대하여 주어진 함수의 결괏값이 true인 요소를 모아 새로운 배열을 반환하는 메서드
- filter함수는 React 코드에서 DB의 자료를 가지고 왔을 때 특정 조건에 해당되는 것만 추려낼 때 쓰는 아주 유용한 함수

- 문법

```
const arr = [5, 1, 3, 2, 6];
```

- 예제

```
const arr = [5, 1, 3, 2, 6];

// odd number
function isOdd(x) {
  if (x % 2) {
    return true;
  } else return false;
}

const output = arr.filter(isOdd);

console.log(output);
```

```
const arr = [5, 1, 3, 2, 6];

const output = arr.filter((x) => x % 2);

console.log(output);
```

  - arr 배열에서 홀수만 뽑아내고 싶을 때
  - x % 2 로직은 짝수면 0을 리턴하기 때문에 0은 자바스크립트에서는 false가 됩니다.

```
const arr = [5, 1, 3, 2, 6];

const output = arr.filter((x) => x % 2 === 0);

console.log(output);
```

  - 짝수만 뽑아내는 filter() 함수

```
let employees = [
  { id: 20, name: "Kim", salary: 30000, dept: "it" },
  { id: 24, name: "Park", salary: 35000, dept: "hr" },
  { id: 56, name: "Son", salary: 32000, dept: "it" },
  { id: 88, name: "Lee", salary: 38000, dept: "hr" },
];
```

```
let departmentList = employees.filter(function (record) {
  return record.dept == "it";
});
console.log(departmentList);
```

- 부서가 it 쪽인 사람만 필터링

reduce

- 배열 각 요소에 대하여 reducer 함수를 실행하고, 배열이 아닌 하나의 결괏값을 반환
- 문법

```
const output = arr.reduce(function (acc, curr) {
  acc = acc + curr;
  return acc;
}, 0);

console.log(output);
```

- 예제

```
const arr = [5, 1, 3, 2, 6];

// sum
function findSum(arr) {
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum = sum + arr[i];
  }
  return sum;
}

console.log(findSum(arr));
```

```
const arr = [5, 1, 3, 2, 6];

// sum
function findMax(arr) {
  let max = 0;
  for (let i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
          max = arr[i];
      }
  }
  return max;
}

console.log(findMax(arr));
```

Rest Operator

arguments

- 파라미터의 개수를 알 수 없는 가변 인자 함수의 경우 사용
- `arguments`는 함수 호출 시 전달된 인수(argument)들의 정보를 담고 있는 객체
- 순회가능한(iterable) 유사 배열 객체(array-like object)이며 함수 내부에서 지역 변수처럼 사용

```
var foo = function () {
  console.log(arguments);
};

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

Spread

- Spread 문법(Spread Syntax, `...`)는 대상을 개별 요소로 분리

```
console.log(...[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

console.log(...'abcdef');  // a b c d e f
console.log(...new Map([['a', '1'], ['b', '2']]));  // [ 'a', '1' ] [ 'b', '2' ]
console.log(...new Set([1, 2, 3]));  // 1 2 3

console.log(...{ a: 1, b: 2 });
// TypeError: Found non-callable @@iterator
```

Rest Parameter

- Rest 파라미터는 Spread 연산자(...)를 사용하여 함수의 파라미터를 작성한 형태를 말한다. 즉, Rest 파라미터를 사용하면 함수의 파라미터로 오는 값들을 "배열"로 전달받을 수 있다.
- 문법

```
function f(a, b, ...theArgs) {
  // ...
}
```

- 예제

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
// expected output: 6

console.log(sum(1, 2, 3, 4));
// expected output: 10
```

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// 콘솔 출력:
// a, one
// b, two
// manyMoreArgs, [three, four, five, six]

myFun("one", "two", "three")

// a, "one"
// b, "two"
// manyMoreArgs, ["three"] <-- 요소가 하나지만 여전히 배열임

myFun("one", "two")

// a, "one"
// b, "two"
// manyMoreArgs, [] <-- 여전히 배열
```

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(element => {
    return multiplier * element
  })
}

let arr = multiply(2, 15, 25, 42)
console.log(arr)  // [30, 50, 84]
```

```
function sortRestArgs(...theArgs) {
  let sortedArgs = theArgs.sort()
  return sortedArgs
}

console.log(sortRestArgs(5, 3, 7, 1)) // 1, 3, 5, 7

function sortArguments() {
  let sortedArgs = arguments.sort()
  return sortedArgs
}

console.log(sortArguments(5, 3, 7, 1))
// TypeError 발생 (arguments.sort is not a function)
```

- 나머지 매개변수와 `arguments` 객체의 차이
  - `arguments` 객체는 **실제 배열이 아닙니다**. 그러나 나머지 매개변수는 `Array` 인스턴스이므로 `sort`, `map`, `forEach`, `pop` 등의 메서드를 직접 적용할 수 있습니다.
  - `arguments` 객체는 `callee` 속성과 같은 추가 기능을 포함합니다.
  - `...restParam`은 후속 매개변수만 배열에 포함하므로 `...restParam` **이전**에 직접 정의한 매개변수는 포함하지 않습니다. 그러나 `arguments` 객체는, `...restParam`의 각 항목까지 더해 모든 매개변수를 포함합니다.

```
* Spread 문법을 사용한 매개변수 정의 (= Rest 파라미터)
   ...rest는 분리된 요소들을 함수 내부에 배열로 전달한다. */
function foo(param, ...rest) {
  console.log(param); // 1
  console.log(rest);  // [ 2, 3 ]
}
foo(1, 2, 3);

/* Spread 문법을 사용한 인수
  배열 인수는 분리되어 순차적으로 매개변수에 할당 */
function bar(x, y, z) {
  console.log(x); // 1
  console.log(y); // 2
  console.log(z); // 3
}

// ...[1, 2, 3]는 [1, 2, 3]을 개별 요소로 분리한다(-> 1, 2, 3)
// spread 문법에 의해 분리된 배열의 요소는 개별적인 인자로서 각각의 매개변수에 전달된다.
bar(...[1, 2, 3]);
```