**단축 평가 논리 계산법**

console.log(true && 'hello'); // hello

console.log(false && 'hello'); // false

console.log('hello' && 'bye'); // bye

// Truthy한 값이면 뒤에 값 표시

console.log(null && 'hello'); // null

console.log(undefined && 'hello'); // undefined

console.log('' && 'hello'); // ""

console.log(0 && 'hello'); // 0

console.log(1 && 'hello'); // hello

console.log(1 && 1); // 1

// &&

// 특정 값이 유효한 경우에만 어떤 값을 조회할 때 사용

const object = null;

const name = object && object.name;

console.log(name); // null

const object = { name : 'Jin' };

const name = object && object.name;

console.log(name); // Jin

// ||

// 어떤 값이 없을 때 다른 값을 사용할 때 사용

const namelessDog = {

name: '',

};

function getName(animal) {

const name = animal && animal.name;

return name || '이름이 없습니다';

}

const name = getName(namelessDog);

console.log(name); // 이름이 없습니다

단축 평가 논리 계산법은 리액트에서 조건부 랜더링할 때에 유용하게 사용할 수 있다.

# 비구조화 할당

const deepObject = {

state : {

information : {

name : 'Jiwon',

languages : ['Korean', 'English']

}

},

value: 5

}

// 비구조화 할당 두 번 하기

const { name, languages } = deepObject.state.information;

const { value } = deepObject;

const extracted = {

name,

languages,

value

};

console.log(extracted);

// 선호 되지 않는 방식

const {

state : {

information : {

name, languages

}

},

value

} = deepObject;

const extracted = {

name,

languages,

value

};

console.log(extracted);

# Class 클래스

### MDN 문서

<https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Classes>

### constructor

class A {}

console.log(new A());

// A{}

class B{

constructor(){

console.log('constructor');

}

}

console.log(new B());

// constructor

// B{}

class C {

constructor(name, age){

console.log('constructor', name, age);

}

}

console.log(new C('Mark',37));

// constructor Mark 37

// C{}

console.log(new C());

// constructor undefined undefined

// c{}

### 멤버변수 - 객체의 프로퍼티

// 1

class A {

constructor(name,age){

this.name=name;

this.age=age;

}

}

console.log(new A('Mark',37)); // A{name:'Mark', age:37}

// 2

class B {

name;

age;

}

console.log(new B()); // 런타임 오류 주의

// 3

class C {

name='no name';

age=0;

constructor (name, age){

this.name=name;

this.age=age;

}

}

console.log(new C('Mark',37));

// {name:'Mark', age:37}

### 멤버함수

class A {

hello1(){

console.log('hello1'.this);

}

hello2=()=>{

console.log('hello2'.this);

};

}

new A().hello1(); // hello1 A{hello2: [function: hello2]}

new A().hello2(); // hello2 A{hello2: [function: hello2]}

class B {

name = 'Mark';

hello(){

console.log('hello', this.name);

}

}

new B().hello(); // hello Mark

### get, set

// 외부에서 값을 바꾸지 않고

// 내부적으로 쓸 경우 \_언더바 사용

class A {

\_name='no name';

get name(){

return this.\_name + '@@@';

}

set name(value){

this.\_name=value + '!!!';

}

}

const a = new A();

console.log(a); // A{ \_name: 'no name' }

a.name = 'Mark'; // set 동작

console.log(a); // A{ \_name: 'Mark!!!' }

console.log(a.name); // Mark!!!@@@

console.log(a.\_name); // Mark!!!

### static 변수, 함수

class A {

static age = 37;

static hello(){

console.log(A.age);

}

}

console.log(A, A.age); // [function:A] {age:37} 37

A.hello(); // 37

class C {

static name = '이 클래스의 이름은 C가 아니다';

}

console.log(C);

// [Function: 이 클래스의 이름은 C가 아니다]

// { name: '이 클래스의 이름은 C가 아니다' }

# Promise

### Promise

비동기 작업을 더 수월하게 해준다.

생성자를 통해서 프로미스 객체를 만들 수 있다.  
생성자의 인자로 executor 라는 함수를 이용한다. resolve(), reject()

생성자를 통해서 프로미스 객체를 만드는 순간, pending 대기 상태  
resolve() 함수를 실행하면 fulfilled 이행 상태  
reject() 함수를 실행하면 rejected 거부 상태

const myPromise = new Promise((resolve, reject) => {

setTimeout(()=>{

resolve('result');

// reject(new Error());

}, 1000)

});

myPromise.then(result=>{

console.log(result); // result

}).catch(e=>{

console.error(e);

})

function increaseAndPrint(n){

return new Promise((resolve, reject)=>{

setTimeout(()=>{

const value= n+1;

if(value===5){

const error = new Error();

error.name = 'valueIsFireError';

reject(error);

return;

}

console.log(value);

resolve(value);

}, 1000);

});

}

increaseAndPrint(0).then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.then(increaseAndPrint)

.catch(e=>{

console.error(e);

})

// 1

function p() {

return new Promise((resolve, reject)=>{

setTimeout(()=>{

resolve();

},1000);

});

}

p().then(()=>{

console.log('1000ms 후에 fulfilled 됩니다.');

});

// 1-1

function p() {

return new Promise((resolve, reject)=>{

setTimeout(()=>{

resolve('hello');

}, 1000);

});

}

p().then((message)=>{

console.log('1000ms 후에 fulfilled 됩니다.', message);

});

// 1000ms 후에 fulfilled 됩니다. hello

// 데이터 넘기는 부분 message 중요!

Promise.resolve(new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve('foo');

},1000);

}).then((data)=>{

console.log('프로미스 객체일 경우, resolve된 결과를 받아서 then이 실행',data);

});

Promise.resolve('bar').then(data=>{

console.log('then 메서드가 없는 경우 fulfilled 됩니다', data);

});

// Promise.all([프로미스 객체들])

function p(ms){

return new Promise((resolve, reject)=>{

setTimeout(()=>{

resolve();

}, ms);

})

}

Promise.all([p(1000),p(2000),p(3000)]).then(()=>{

console.log('모두 fulfilled된 이후에 실행');

})

# async/await

또 다른 비동기를 다루는 방식  
async function 함수이름(){}  
또는  
const 함수이름 = async () => {}

// index.js

function sleep(ms){

return new Promise(resolve => setTimeout(resolve,ms));

}

// async/await

async function process(){

console.log('안녕하세요!');

await sleep(1000);

console.log('반갑습니다.');

return true;

}

process().then(value => {

console.log(value);

});

// 안녕하세요

// 반갑습니다

// true

// error catch

async function makeError(){

await sleep(1000);

const error = new Error();

throw error;

}

async function process(){

try {

await makeError();

} catch(e) {

console.log(e);

}

}

process();

// 1초 뒤에 Error 발생

// 바로 실행

(async function process(){

try {

await makeError();

} catch(e) {

console.log(e);

} finally {

console.log('end');

}

})();

### async function에서 return되는 값을 Promise.resolve 함수로 감싸서 리턴

function p(ms){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve(ms);

// reject(new Error('reason');

}, ms);

});

}

async function asyncP(){

const ms = await p(1000);

return "Mark: " + ms;

}

(async function main(){

try{

const name = await asyncP();

console.log(name);

} catch(error){

console.log(error);

})();

// resolve → Mark : 1000ms

// reject → Error

### 연속된 promise와 async await

function p(ms){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve(ms);

// reject(new Error('reason');

}, ms);

});

}

// Promise

p(1000).then(()=>p(1000))

.then(()=>p(1000))

.then(()=>{

console.log('3000ms 후 실행');

});

// async await

(async function main(){

await p(1000);

await p(1000);

await p(1000);

console.log('3000ms 후 실행');

})();

### Promise.all과 Promise.race

function p(ms){

return new Promise((resolve,reject)=>{

setTimeout(()=>{

resolve(ms);

// reject(new Error('reason');

}, ms);

});

}

// Promise.all

(async function main(){

const results = await Promise.all([p(1000),p(2000),p(3000)]);

console.log(results);

})(); // [1000,2000,3000]

// Promise.race

(async function main(){

const result = await Promise.race([p(1000),p(2000),p(3000)]);

console.log(result);

})(); // 1000