## 호이스팅

자바스크립트는 선언문들을 맨 위로 끌어올린다.

console.log(a());

console.log(b());

console.log(c());

function a() {

return 'a'

}

var b = function bb(){

return 'bb'

}

var c = function (){

return 'c';

}

이 코드는 첫줄부터 오류가 나야 될 것 같지만 2번째 줄부터 오류가 난다. 왜냐하면 호이스팅으로 인해 선언 부분이 맨위로 올라가기 때문이다.

그래서 사실은

function a() {

return 'a'

}

var b;

var c;

console.log(a());

console.log(b());

console.log(c());

b = function bb(){

return 'bb'

}

c = function (){

return 'c';

}

이런 코드가 되는 것이다. 호이스팅은 선언과 관련이 있지 할당과는 관련이 없기 때문에 b부터는 오류가 나게 된다.

## 함수선언문과 표현식의 차이점.

function a() { //함수 선언문 function declaration

return 'a'

}

var b = function bb(){ //기명함수 표현식 named function expression

return 'bb'

}

var c = function (){ //익명함수표현식(unnamed / anonymous)function expression

return 'c';

}

익명함수 표현식이 선언되고 진행되는 방식

변수 c선언 , 익명함수 선언, 변수 c에 익명함수를 할당,

선언문과 표현식의 차이는 할당에 있음. 선언문을 하게 되면 함수 자체가 호이스팅이 되어 맨위로 올라가지만 표현식은 변수만 호이스팅 되어 올라가고 할당되는 함수는 그대로 남아있게된다.

호이스팅이 되는가에 따라 실무에서 엄청난 차이가 남.

만약에 협업을 할 때 저 위에 내가 선언한 sum과 저 밑에 다른 sum이 새로 선언이 되면 함수 호이스팅이 되면서 위에 선언이 되게 된다. 그래서 앞의 sum은 사라지고 밑에 sum만 선언이 되면서 모두 마지막에 선언한 sum으로 사용하게 된다. 그래서 여러가지 오류가 나게 된다.

이런 경우가 아니더라도 호이스팅은 코드는 위에서부터 아래로 실행되어야 한다는 지극히 인간적인 관점에서 벗어난다. 가독성 면에서 좋지 않음. 혼자 개발 할 때는 어디서나 적고 실행해도 된다는 장점이 있겠지만 다른 사람이 볼 때는 코드를 이해하는데 상당한 저해가 될것이다.

함수 선언문 보다는 무조건 함수 표현식을 써라!

es6부터는 function이라는 단어 자체를 쓰지 않고 화살표 함수로 한다.

## 함수 스코프와 실행 컨텍스트

스코프 : 변수의 유효범위

실행 컨텍스트 : 실행되는 코드덩어리(추상적 개념)

소크프는 정의될 때 결정된다. / 실행 컨텍스트는 실행 될 때 생성된다.

실행 컨텍스트에는 호이스팅, this 바인딩 등의 정보가 담긴다.

사용자가 함수를 호출 햇을 때 외부적으로 함수를 실행하기 위해 필요한 정보들을 불러 모아놓은 집합

var a = 1;

function outer() {

console.log(a);

function inner() {

console.log(a);

var a = 3;

}

inner();

console.log(a);

}

outer();

console.log(a);

전역컨텍스트 . GLOBAL

var a =1 이 처음 전역 컨텍스트로 생성

outer 함수 호출 -> outer 컨텍스트 생성

함수 호이스팅으로 인해 먼저 함수 inner 선언

그리고 console.log(a) outer에서 찾아도 a가 없으니 전역에서 찾아서 a =1

inner 함수 호출 -> inner 실행컨텍스트 생성

var a; 가 선언

console.log(a)가 undefined로 됨 그리고 a=3 이 할당됨

그리고 console.log(a)를 하면 outer에 없으니 1이 출력되고

마지막도 global에서 탐색해서 1이 출력되고 전역 실행 컨텍스트가 종료된다.

## method (메소드)

var obj = {

a: 1,

b: function bb() {

console.log(this);

},

c: function() {

console.log(this.a);

}

};

obj.b();

obj.c();

console.dir(obj.b);

console.dir(obj.c);

메소드는 .이 붙은 함수이다. 그니깐 obj.b(), obj.c() 들을 의미한다. 함수와 메소드의 차이는 this 가 바인딩 되느냐 아니냐이다. 그래서 b의 this를 출력하면 obj가 나온다. c를 출력하면 obj.a 가나온다. this가 바인딩 되었기 때문이다.

## 콜 백 함수

콜 부르다 백 뒤로..

something will call this function back sometime somehow

어떤 것이 이 이함수를 호출할거야 언젠가 어떻게든 돌려준다..

무언가가 이 함수를 나에게 다시 돌려줄거야 호출해서 언젠가 어떻게든

무언가가 이 콜백 함수를 언젠가 어떻게든 실행해서 나에게 다시 돌려줄거야.

제어권을 넘겨준다. 맡긴다… 맡기고자 하는 대상한테

가장 간단한 콜백.

setInterval(function () { // 주기함수 호출 / 인자1: 콜백함수

console.log('1초마다 실행될겁니다.');

}, 1000);// 인자2 주기(ms)

제어권을 setInterval 에게 넘겨준다.

setInterval( callback, milliseconds) setInerval 함수는 milliseconds 시간마다 callback 함수를 호출하는 함수. 원래부터 그렇게 정의가 되어있음.

var arr = [1, 2, 3, 4, 5];

var entries = [];

// arr, forEach 호출 // 인자1 콜백함수

arr.forEach(function(v, i) {

entries.push([i, v, this[i]])

}, [10,20,30,40,50]) // 인자2 this로 인식할 대상(생략가능)

console.log(entries);

//결과 [[0,1,10],[1,2,20],[2,3,30],[3,4,40],[4,5,50]]

forEach() 메소드는 배열 요소마다 한 번씩 제공한 함수를 실행합니다.

arr.forEach(callback, [thisArg])

callback 함수에는 순서대로 currentValue 배열에서 현재처리 중인 요소, index 배열에서 현재 처리중인 요소의 인덱스 array forEach() 가 적용되고 있는 배열.

thisArg, 선택 사항 . callback을 실행할 때 this로서 사용되는 값.

파라미터의 순서는 규칙 그대로. 변수명과는 상관없음.

콜백함수의 특징

다른 함수(A)의 매개변수로 콜백함수(B)를 전달하면 A가 B의 제어권을 갖게 된다.

특별한 요청(bind)이 없는한 A에 미리 정해진 방식에 따라 B를 호출한다.

미리 정해진 방식이란 this에 무엇을 바인딩할지, 매개변수에는 어떤 값들을 지정할지 어떤 타이밍에 콜백을 호출할지 등.

주의 ! 콜백은 함수다 메소드 x 메소드로 주면 에러..