# 190627_thu_

| | |
|---|---|
| Last Edited | @Jun 27, 2019 8:08 PM |
| Property | |
| Tags | this |

## this keyword

- code가 실행되면 먼저

-Global memory 생성되고

```
var num = 2;
function pow(num) {
  return num * num;
}
```

-Global execution context 가 생성된다.

- 그러고 function이 호출되면

-Local meory 가 생성되고

-Local execution context 가 생성된다.

*execution context : 실행 컨텍스트

-어떤 함수가 호출되면 실행 컨텍스트가 만들어진다.

-이거는 call stack에 push,

-함수를 벗어나면 call stack에서 pop

-scope 별로 생성된다

-여기에 담기는 것 : - scope 내 변수 및 함수(Local, Global)

- 전달인자(arguments)

- 호출된 근원(caller)

- this

-콘솔열어서 소스창을 보면 call Stack에서 execution context를 볼수있다. (함수들)

그리고 그 아래 Scope에서 변수, 함수목록과 this 확인가능

# this

는 모든 함수 scope내에서 자동으로 설정되는 특수한 식별자,

execution context의 구성요소 중 하나로, 함수가 실행되는 동안 이용할 수 있다.

- 5 patterns of Binding 'this' // 디스가 결정되는 패턴

1. Global: window
2. Funtion 호출: window
3. Method 호출: 부모 object
4. Construction mode (new 연산자로 생성된 function영역의 this): 새로 생성된 객체
5. .call or .apply 호출: call, apply의 첫번째 인자로 명시 된 객체

ex) 1.Global: window 2.function 호출:window

```
var name = 'Global Variable';
console.log(this.name); //"Global Variable"

function foo() {
  console.log(this.name); //"Global Variable" //여기서 this는 window
}

foo();
```

```
var name = 'Global Variable';

function outer() {
  function inner() {
    console.log(this.name); //"Global Variable"
  }
  inner();
}
outer();
//일종의 클로전데 여기도 this는 window가 된다.
//펑션인보케이션, 글로벌 인보케이션에서 this === window 가 된다.
```

```
var name = 'Global Variable';

function outer2() {
  var closure = function() {
    console.log(this.name); //"Global Variable"
  }
  return closure;
}

outer2()();
```

ex) Method 호출: 부모 object

*method와 functiond의 차이점 : 객체 안에 있는 함수가 method

```
var counter = {
  val: 0,
  increment: function(){
    this.val += 1;   // this === counter
  }
};

counter.increment();
console.log(counter.val); // 1
counter['increment']();
console.log(counter.val); // 2
```

```
var obj = {
  fn: function(a,b) {
    return this;
  }
};
var obj2 = {
  method: obj.fn
};

console.log(obj2.method() === obj2); // true
console.log(obj.fn() === obj); // true
//앞쪽(부모)이 뭐냐에 따라서 결정된다
//왜냐면 실행시키는 시점(콜타임)에서 부모를 찾기때문

let obj4 = {
  key1: {
    key2: function() { console.log(this) }
  }
}
obj4.key1.key2(); //key1의 값 , 바로 윗 부모
```

ex) Construction mode (new 연산자로 생성된 function영역의 this): 새로 생성된 객체

```
function F(v) {
  this.val = v;
}

//create new instance of F
var f = new F('WooHoo!');  //여기서 this 는 f

console.log(f.val); // WooHoo!
console.log(val); // ReferenceError
```

ex) 5. .call or .apply 호출: call, apply의 첫번째 인자로 명시된 객체

```
function identify() {
  return this.name.toUpperCase();
}
function speak() {
  var greeting = "Hello, I'm " + identify.call(this);
  console.log(greeting);
}
var me = { name: "sui" };
var you = { name: "woo" };

identify.call(me); // SUI
identify.call(you); // WOO
speak.call(me); // Hello, I'm SUI
speak.call(you); // Hello, I'm WOO
```

```
var add = function (x, y) {
  this.val = x + y;
}
var obj = {
  val = 0;
};

add.apply(obj, [2, 8]};
console.log(obj.val); // 10
add.call(obj, 2, 8);
console.log(obj.val); // 1
```

---

<세션영상 다시보기>

```
//이거 이해가 안되고있음 190627

var fns = [];
for (var i=0; i<3; i++) {
  fns[i] = function() {
    console.log('My value: ' + i);
  }
}
for (var j=0; j<3; j++) {
  fns[j]();
}

console.log(
```

var을 let으로 바꿔서 해결이 되는 문제

var일때 scope는 2개이다. 전체가 하나, 펑션있는거 하나

let으로 했을때는 모든 변수가 펑션 블락안에 같힌다. 그리고 i는 상위 블락스코프에서 찾게된다.

이 i가 1,2,3인데 let으로 하면 그 블락안에서 for가 딱 돌지 근데

var로 하면 i는 for문을 다 돌고나서 전체 스코프에서 3인체로 남아있게 된다. 그래서 let을 써야해

*for문⇒ block scope

this 본격적으로 들어가보자

지난번에 class instance만들때 this봤는데 이때,  this는 new키워드를 가지고 만드는 인스턴스가 this가 된다.

EXECUTION CONTEXT  의 개념을 알아보자

-코드가 만들어질때 메모리가 생기는데(메모리 테이블 연상), 이 만들어진 (전역변수, 전역함수)  변수와 변수의 값을 익스큐전 컨텍스트에 저장이 된다라고 예기한다.

-실행컨텍스트 : 펑션스코프마다 생긴다

-지역변수도 마찬가지로 스코프에 따라서 실행 컨텍스트가 생긴다. 스코프의 수와 실행컨텍스트의 수는 같다.

-모든 함수 scope내에서 자동으로 설정되는 특수한 식별자

-execution context의 구성요소 중 하나로, 함수가 실행되는 동안 이용할 수 있다.

-end