- 함수, scope, closure

1. Javascript 함수

❖"Javascript 함수는 일급 객체"

- 다른 언어(Java, C#)의 메서드와는 다르다.

❖일급 객체란?

- 함수는 Object 타입의 인스턴스이다.
- 변수에 함수를 저장할 수 있다.
- 다른 함수의 파라미터로 함수를 전달할 수 있다.
- 함수가 다른 함수를 리턴할 수 있다.
- 함수가 자료구조(data structure)에 포함될 수 있어야 한다.

❖한마디로 **자바스크립트 함수는 객체이다.**

❖객체로서의 특징을 가진 함수를 정확하게 이해해야 함.

2. 일급객체로서의 함수 특징

❖함수의 인자로 다른 함수를 전달할 수 있음

```
//setTimeout 함수의 첫번째 인자 : 함수
//함수를 인자로 전달했음
setTimeout( ( ) => {
    ......
}, 2000)
```

❖함수는 함수를 리턴할 수 있음

```
const setMessage = (message) => {
    return (name) => {
        return message + " " + name;
    }
}

const m = setMessage("안녕");
m("홍길동");
m("이몽룡");
m("박문수");
```

3. 호이스팅(Hoisting)

❖ 호이스팅이란?

- hoist : 들어올리다
- 물건 따위를 사용할 수 있도록 필요한 위치(높은 곳)로 들어올린다는 의미
- 실행에 필요한 함수를 미리 생성하고, 변수를 위한 메모리 할당을 미리 한다는 것
  - JS코드가 실행되면 가장 먼저 전역 실행컨텍스트가 생성됨.
  - 그 후 선언적 방식의 함수가 생성되고, var 키워드가 사용된 변수가 미리 만들어짐
  - 그 이후에 한줄씩 코드를 실행해 나감
  - Hoisting --〉 Execution
- 호이스팅(O), 호이스팅(X)

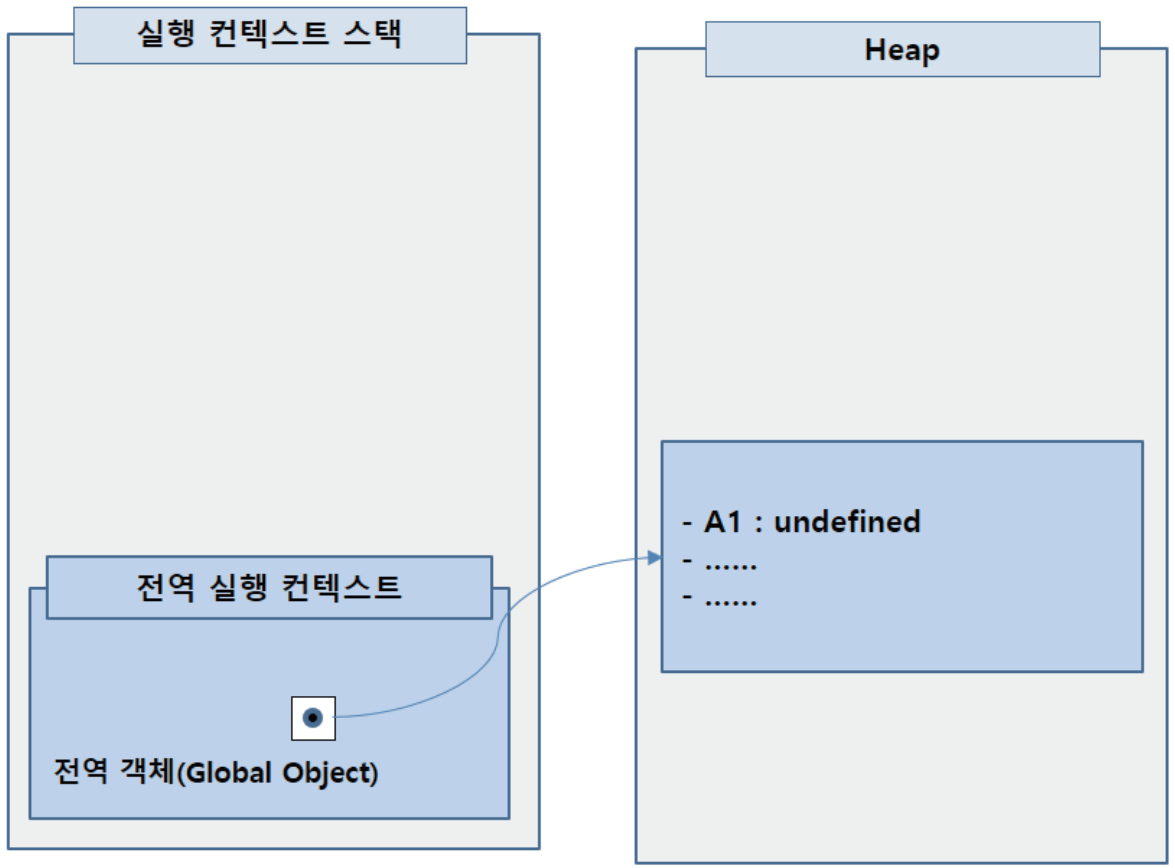
```
//호이스팅 하므로 오류(X)
console.log(A1);
var A1 = "hello";

console.log(A());
function A() {
  console.log("World");
}
```

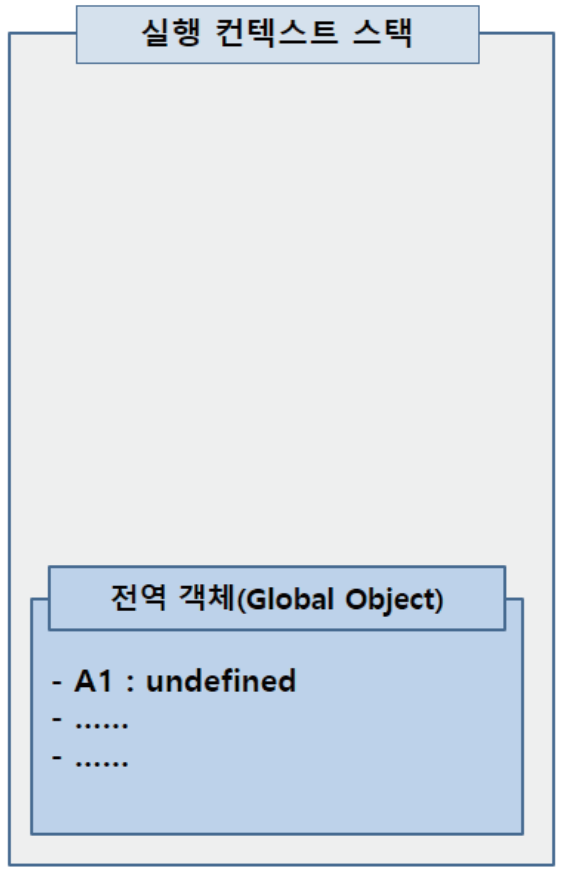
```
//호이스팅 하지 않으므로 오류 발생
console.log(A1);
let A1 = "hello";

console.log(A());
const A = () => {
  console.log("World");
}
```

3. 호이스팅(Hoisting)

❖전역 실행 컨텍스트와 전역 객체

- 편의상 오른쪽과 같이 표현할 것임





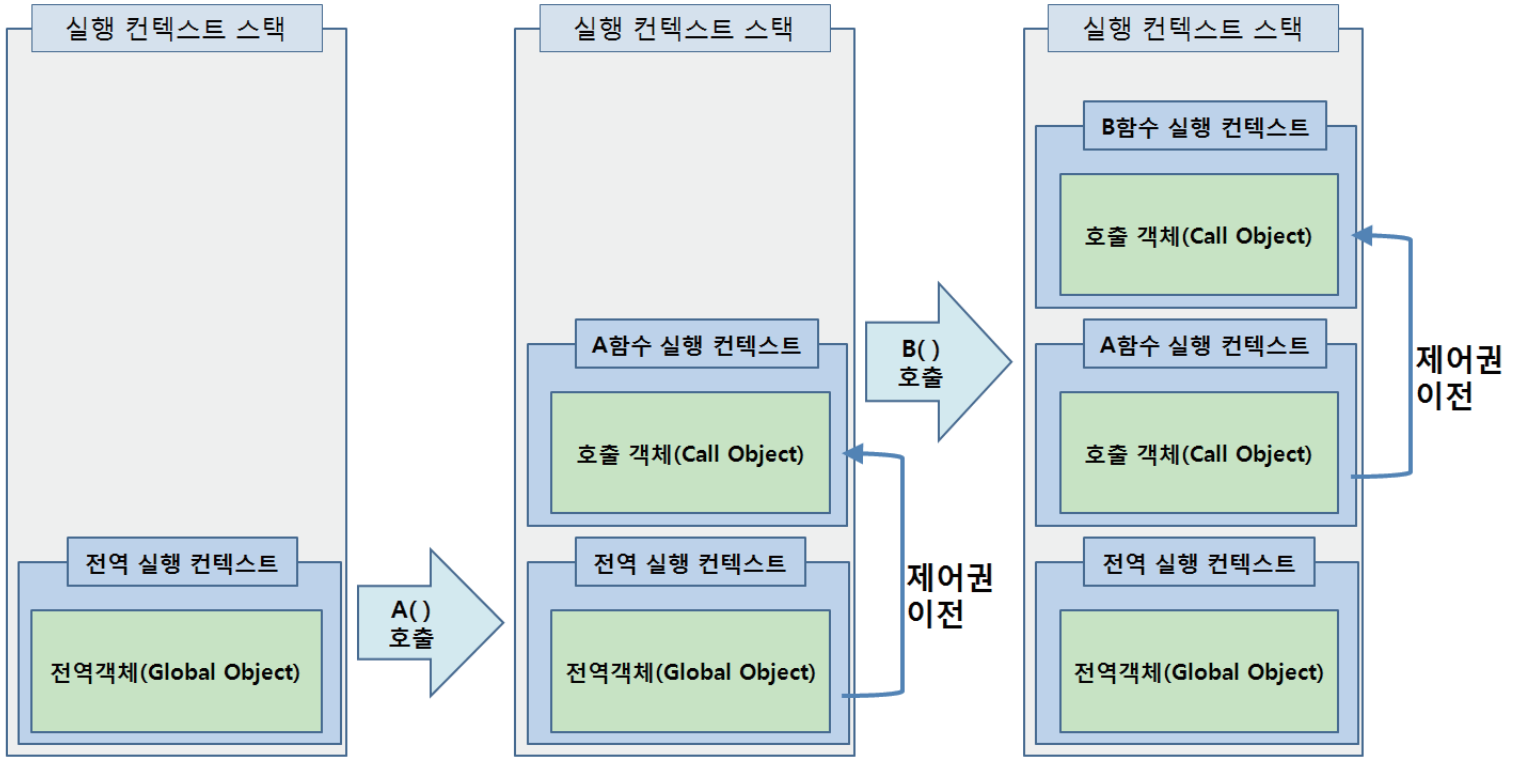
4. 함수의 호출과정

❖실행 컨텍스트

- "자바스크립트 코드가 실행되는 환경"
- 전역 실행 컨텍스트가 만들어지고 난 뒤, 실행 컨텍스트 내부에 전역 객체(Global Object)를 생성하여 실행에 필요한 값들을 저장한다.
- "전역 객체의 속성" = "전역 변수"

❖샘플 코드

```
const B = () => {
    console.log("hello");
}
const A = () => {
    B();
    console.log("world");
}
A();
```



4. 함수의 호출과정

❖함수의 호출 과정

- 함수 실행을 위한 실행 컨텍스트를 생성하여 실행 컨텍스트 스택에 추가한다.
- 실행 컨텍스트 안에 호출 객체(Call Object)를 생성한다.
- 호출 객체 안에 함수 호출시에 전달되는 파라미터 값과 arguments 객체를 생성한다.
- 스코프 체인(Scope Chain) 정보를 생성한다.
- 호출 객체 안에 선언적 방식의 내부 함수를 미리 만든다.(호이스팅)
- 호출 객체 안에 var 키워드로 선언된 변수를 미리 만든다(호이스팅)
- this를 연결한다(this binding)
- 함수 내부의 코드를 실행한다.
  - 호이스팅되지 않는 변수, 함수는 코드가 실행될 때 만들어짐

❖스코프 체인은 리스트 형태의 구조

- 현재 호출 중인 함수가 정의된 호출 객체, 전역 객체를 가리키는 정보를 가지고 있음

4. 함수의 호출과정

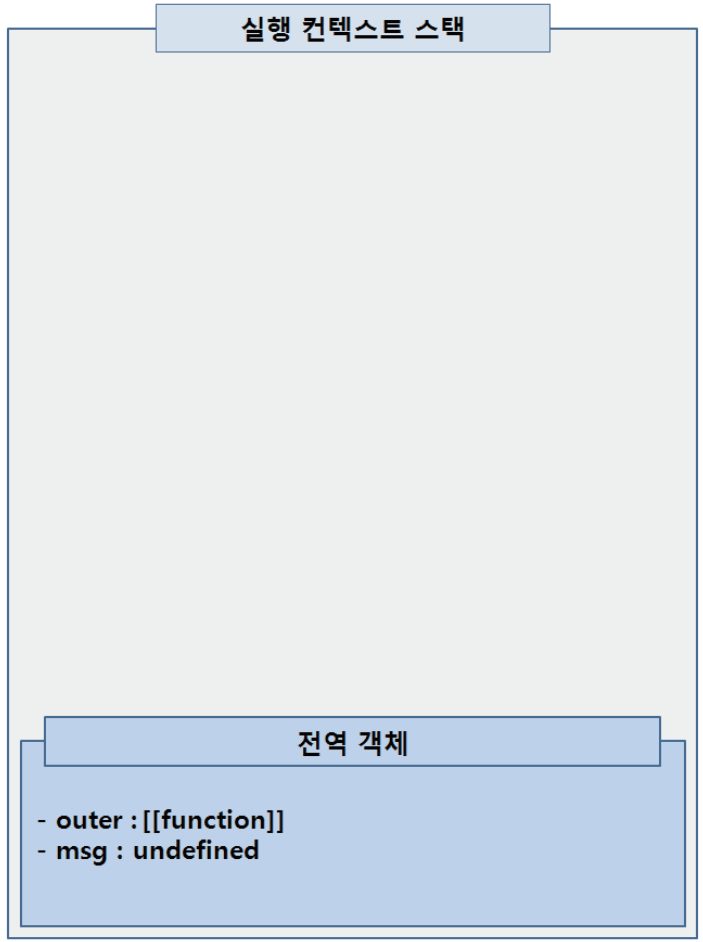
❖다음 예제를 통해 실행 과정을 살펴보자

```
var msg = "GLOBAL";
function outer() {
   var msg = "OUTER";
   console.log(msg);
   function inner() {
       var msg = "INNER";
       console.log(msg);
   }
}

outer();
```

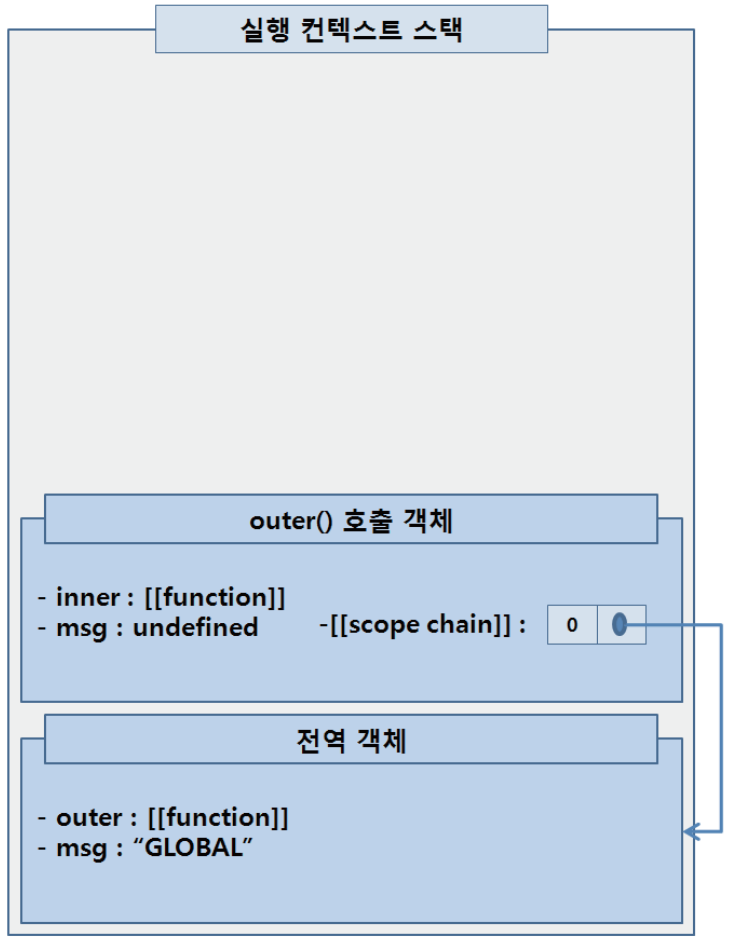
```
** 실행 결과
OUTER
INNER
```

4. 함수의 호출과정

- 1단계

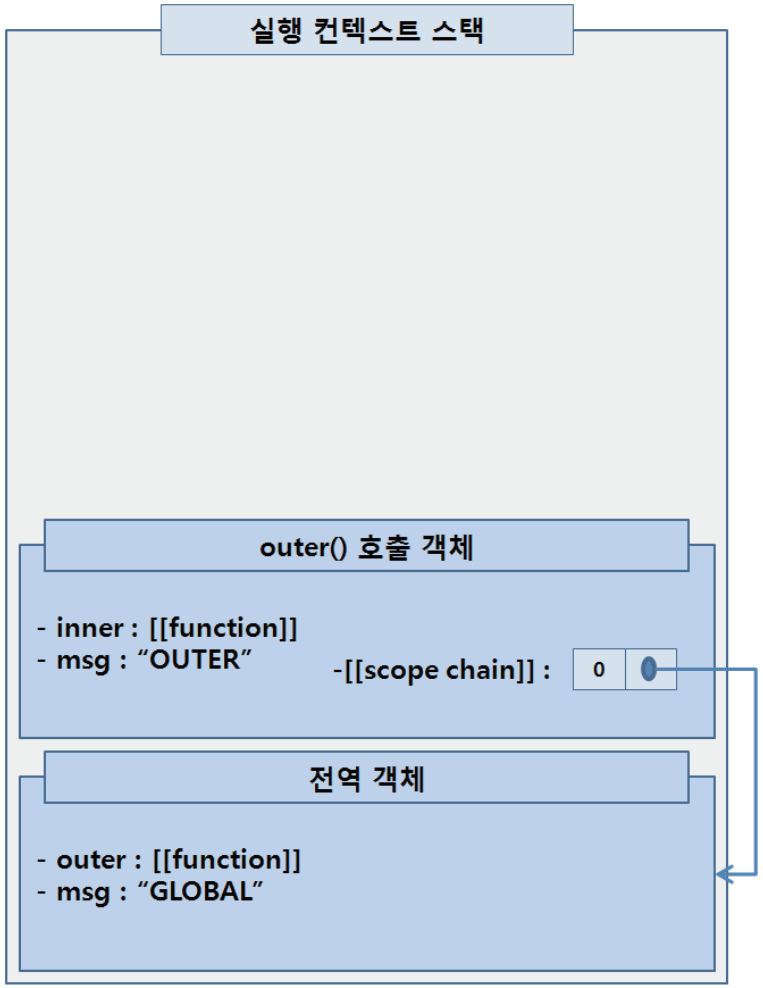


- 2단계

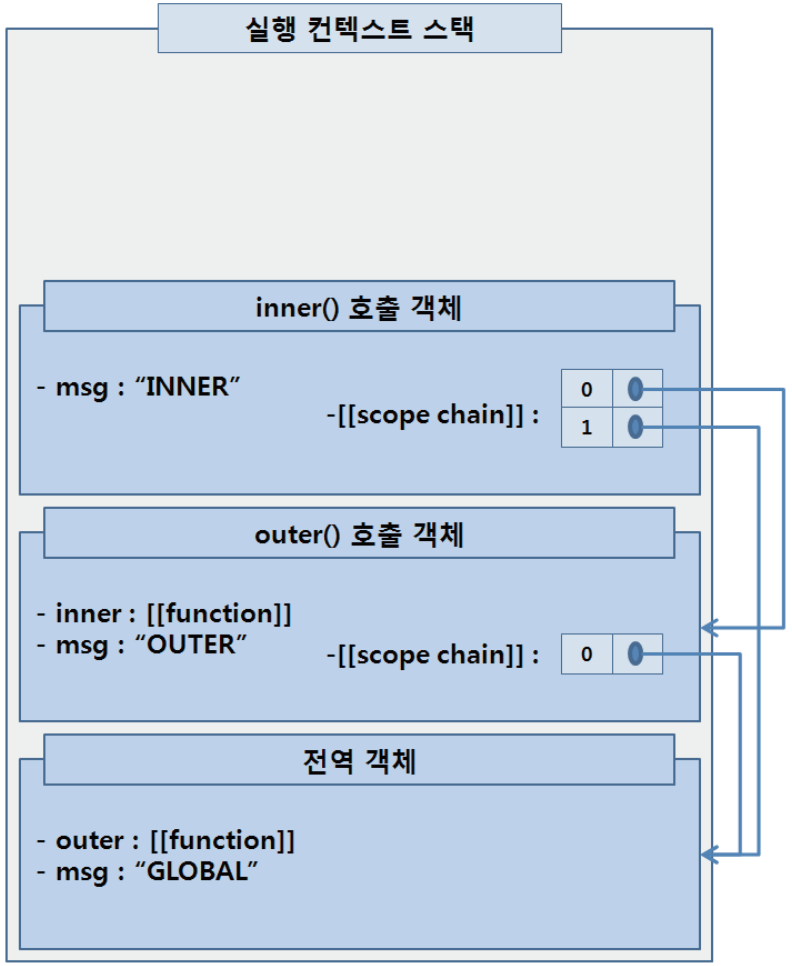


4. 함수의 호출과정

- 3단계



- 4단계



4. 함수의 호출과정

- 5단계
  - 함수 호출이 완료되면 각각의 실행 컨텍스트는 스택에서 제거되고
  - 실행 제어권을 스택 상의 아래에 있는 실행 컨텍스트로 넘겨준다.
  - 그 결과 실행 컨텍스트가 참조하고 있던 호출 객체는 가비지 컬렉션 대상이 되어 메모리가 회수되는 절차를 밟게 된다

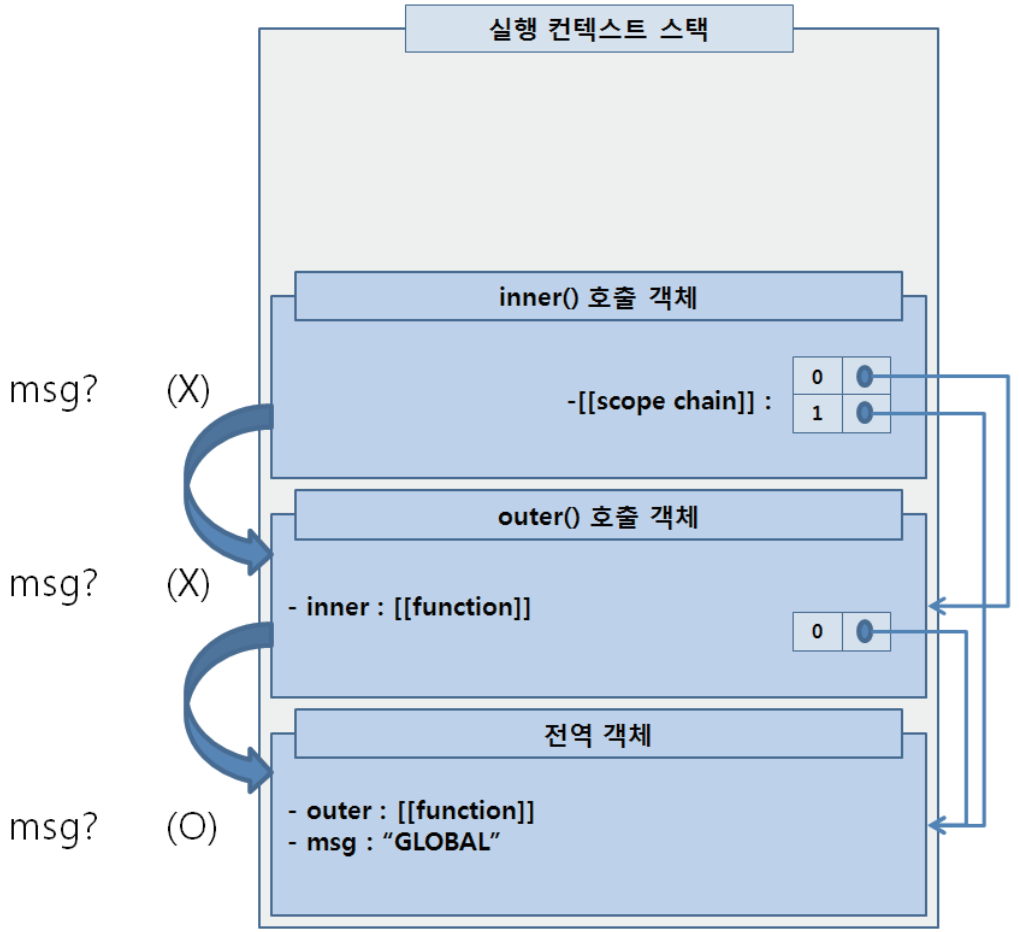
5. 스코프와 스코프 체인

❖호출 객체 단위로 스코프가 결정된다

❖호출 객체 내에 스코프 체인에 대한 정보를 가지고 있다

```
var msg = "GLOBAL";
function outer() {
    //var msg = "OUTER";
    console.log(msg);
    function inner() {
        //var msg = "INNER";
        console.log(msg);
    }
}

outer();
```

```
** 실행 결과
GLOBAL
GLOBAL
```



5. 스코프와 스코프 체인

❖호출 객체 단위로 스코프가 결정된다.

- 2행의 num과 3행의 num은 동일한 변수를 가리킨다.
- var 키워드를 사용하면 블록 단위 스코프는 존재하지 않음
  - ES6의 let 키워드를 사용하면 블록 단위 스코프를 적용할 수 있음

```
const test = () => {
   var num = 100;
   for (var num=0; num < 10; num++) {
       console.log(num);
   }
   return num;
}

var result = test();
console.log("최종 num : " + result);

//num 변수를 var 대신 let으로 선언하고도 실행해보자
```

```
** var 일때 결과
0
1
2
3
4
5
6
7
8
9
최종 num : 10
```

```
** let 일때 결과
0
1
2
3
4
5
6
7
8
9
최종 num : 100
```

6. 스코프 연습 문제

❖문제 1

```
01: //----<1번>
02: function test1(a1) {
03:     a1();
04:     function a1() {
05:         console.log("world");
06:     }
07: }
08: test1(function() { console.log("hello"); })
```

- 호이스팅 단계에서
  - arguments 및 파라미터 전달에서 2행의 a1으로 8행의 익명함수 전달
  - 4행의 선언적 함수 값이 a1에 할당되면서 a1이 변경됨
  - 호이스팅 단계가 완료되고나서 함수 내부 코드 실행 --〉 a1() 함수 호출
  - 따라서 결과는 "world"

6. 스코프 연습 문제

❖문제 2

- 핵심 포인트는 13행의 a2

```
11: var a2 = 1;
12: function test2() {
13:   a2 = 10;
14:   return;
15:   function a2() {}
16: }
17: test2();
18: console.log(a2);
```

12행 test2( ) 함수 생성

11행 전역 변수 a2 정의

17행 test2( ) 호출로 호출 객체 생성

15행 호이스팅 단계로 a2 내부 함수 생성

13행 a2 함수 변수에 10을 할당(데이터 타입이 function에서 number로 변경)

14행 return문 실행으로 함수 실행 종료

18행 a2 변수 출력(기존 전역 변수값 10이 그대로 유지되었음)

6. 스코프 연습 문제

❖문제 3

```
21: var test3 = (function f(){
22:     function f(){ return "hello"; }
23:     return f;
24:     function f(){ return "world"; }
25: })();
26: console.log(test3());
```

- 즉시 실행함수 호출로 인해 만들어진 호출객체 내부에서 호이스팅 단계가 일어나고 22행, 24행의 선언적 함수가 순차적으로 만들어진다. 호이스팅이 완료되고 나면 23행이 실행되면서 리턴한다.
- 리턴된 값은 21행의 test3 변수에 할당된다. 따라서 test3() 호출 결과는 world
- 즉시 실행 함수
  - 즉시 실행 함수(IIFE:Immediately Invoked Function Expression)는 만들어진 직후에 바로 호출되는 함수를 말한다. 바로 호출되므로 익명 함수(Anonymous function)를 이용한다.
  - 이름이 없는 함수이긴 하지만 호출되므로 독립적인 호출 객체를 만들기때문에 별도의 스코프를 가진다.
  - (function( ) { })( );

7. 클로저

❖ 클로저란?

- 정의
  - 외부 함수 내의 내부 함수가 전역에서 참조되고, 내부 함수가 외부 함수 내의 지역 변수를 이용할 수 있게되는 현상 또는 내부 함수
  - 스코프 체인을 이용해 호출이 완료된 함수의 내부 변수를 참조할 수 있는 방법
  - 내부 함수를 통해 외부 함수의 실행 컨텍스트 정보를 접근할 수 있는 방법
  - 특정 함수 내의 지역 변수를 외부에서 접근할 수 있도록 하는 내부 함수

- 서로 다른 설명처럼 보이지만 동일한 내용임
- 클로저를 이해하려면 함수호출과정, 스코프, 호이스팅을 이해하면 됨.
  - 클로저의 내용은 전혀 새로울게 없음

7. 클로저

❖호출객체가 가비지 컬렉션 되지 않는 경우

- 함수 호출이 완료되면 호출 객체는 가비지 컬렉션 대상이 되지만 그렇지 않은 경우도 있음
- 함수 안에 내부 함수가 정의되고, 그 내부 함수를 전역에서 접근할 수 있는 상황
  - 내부 함수가 리턴되는 경우
  - 내부 함수가 전역 변수에 할당되는 경우

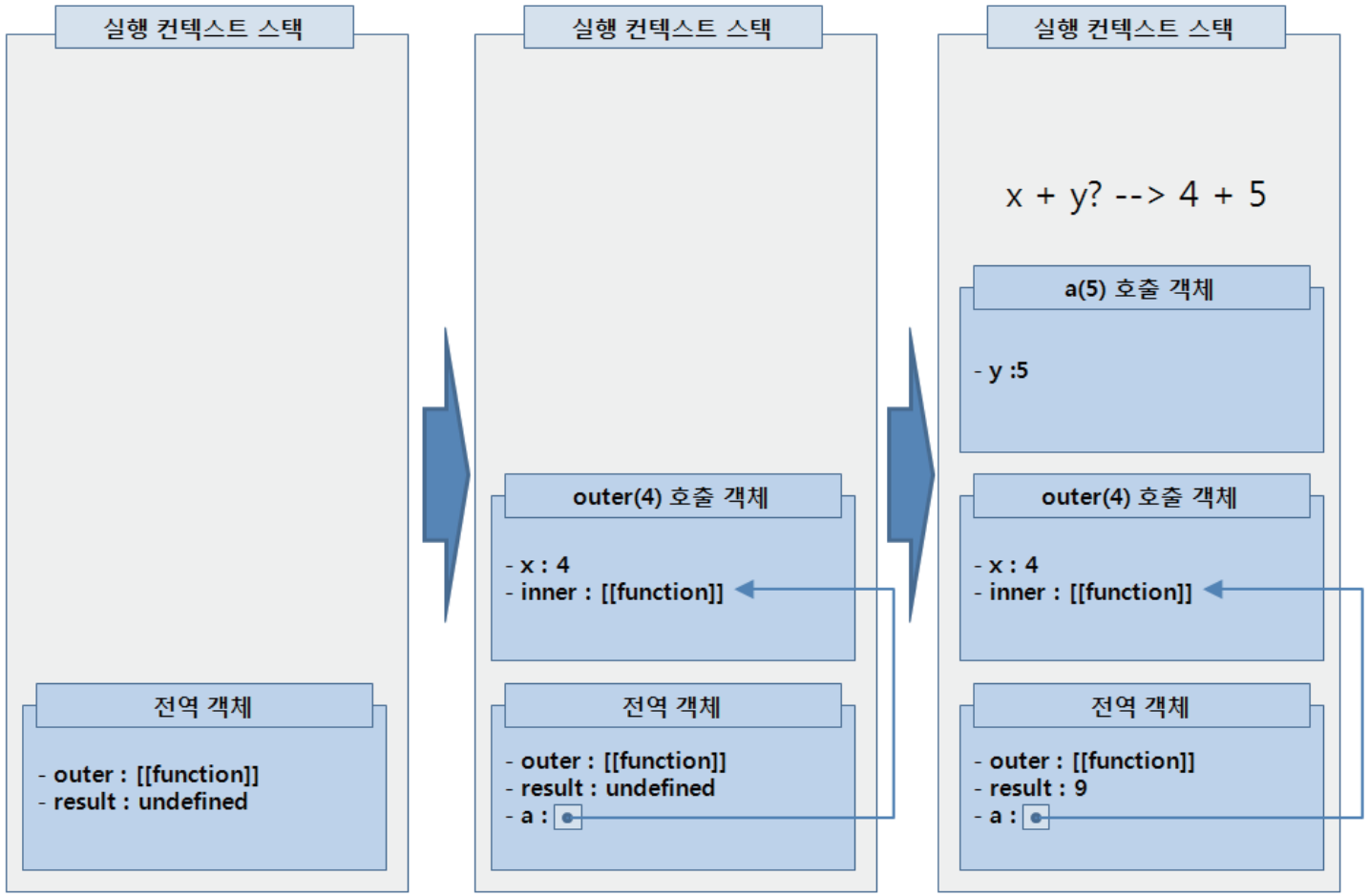
```
const outer = (x) => {
   const inner = (y) => x+y;
    return inner;
}

let a = outer(4);
let result = a(5);
console.log(result);                    //결과 : 9
```

```
const outer = (global, x) => {
   const inner = (y) => x+y;
    global.a = inner;
}

outer(window, 4);
let result = a(5);
console.log(result);                    //결과 : 9
```

7. 클로저

❖ 이전 페이지 예제 실행 흐름

- 클로저 함수 내부의 자유변수는 오로지 클로저 함수를 통해서만 접근이 가능함




```
function outer(x) {
    function inner(y) {
        return x+y;
    }
    return inner;
}
var a = outer(4);
var result = a(5);
console.log(result);
```

자유 변수

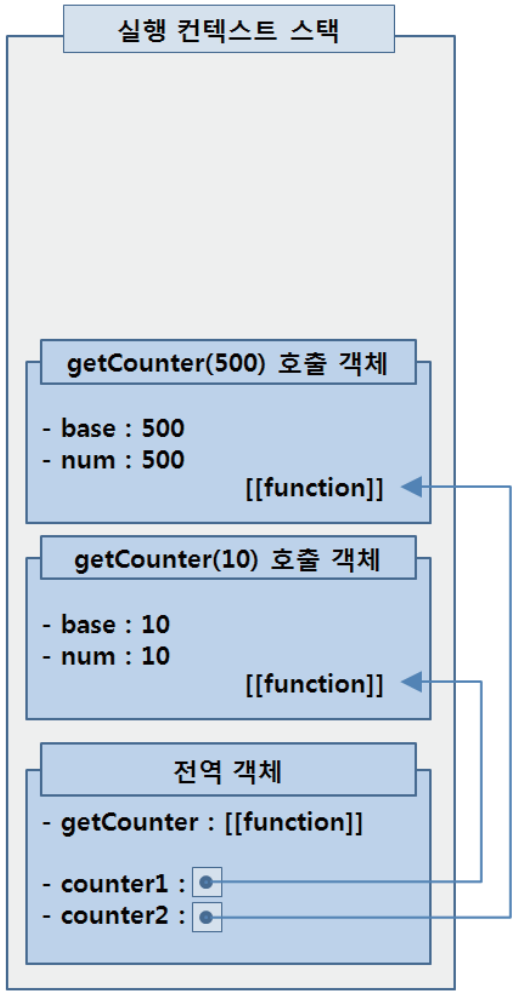
클로저 함수

7. 클로저

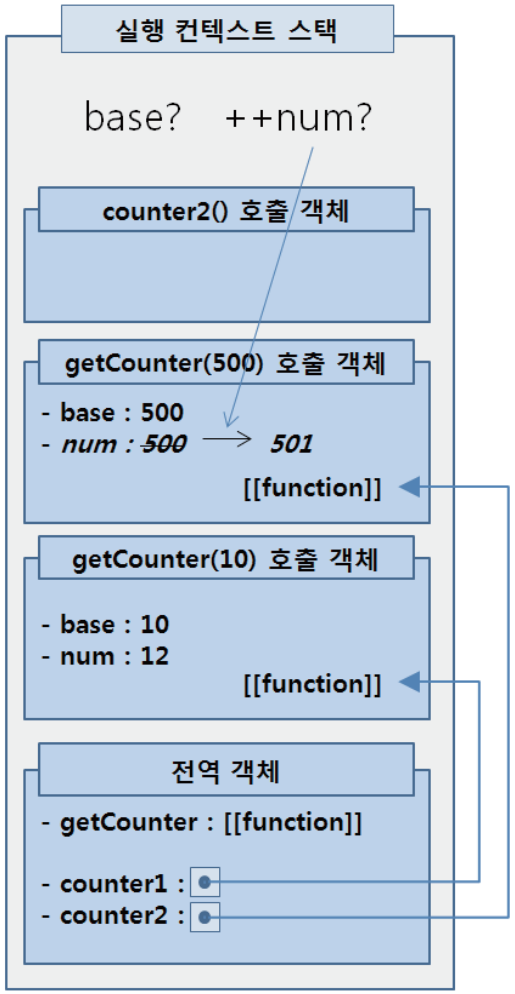
❖클로저가 여러개 생성되는 경우

- 단지 함수의 중첩구조만 고려해서는 안됨

```
const getCounter = (base) => {
   let num = base;
   return ()=> {
      return { base, count: num++ };
   }
}

let counter1 = getCounter(10);
let counter2 = getCounter(500);

// 여기까지 A 참조, 이후는 B 참조
console.log(counter1());
console.log(counter1());
console.log(counter2());
console.log(counter2());
console.log(counter2());
```



<A>



<B>

8. 클로저와 리액트 훅

❖ 6회차 때의 useFetch 훅

```
……(생략)
const useFetch = (url, params) => {
  const [response, setResponse] = useState();
  const [error, setError] = useState();
  const [isLoading, setIsLoading] = useState(false);

  const fetchData = async () => {
    ……(생략)
  };
  const requestData = () => {
    fetchData();
  };
  return { response, error, isLoading, requestData };
};
export { useFetch };

-------------------------------------------------------
//App 컴포넌트에서 호출할 때
const { response, isLoading, error, requestData } =
          useFetch(`/todolist_long/${owner}`, { timeout: 5000 });
```