함수란?

- 많은 개발자들이 오해하고 있음
- 함수는 자바의 메서드와는 다름
- 일급 객체
  - 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.

함수의 정의(1)

함수의 정의 방법

[예제 03-01 : 함수 생성 방식]

```
//1. Function 생성자 함수 방식
var f1 = new Function("x", "y", "return x+y");
console.log(f1(7,5));   //결과 : 12

//2. 선언적 함수 방식
function f2(x,y) {
    return x-y;
}
console.log(f2(7.5));   //결과 : 2

//3. 함수 리터럴 방식
var f3 = function(x,y) {
    return x*y;
}
console.log(f2(7.5));  //결과 : 35
```

- Function 생성자 함수 방식은 클로저를 생성할 수 없으므로 거의 사용하지 않음

함수의 정의(2)

호이스팅의 순서

- 선언적 함수 방식으로 생성된 함수
- var 키워드로 선언된 변수
  - 선언적 함수와 var의 차이가 있다면, 선언적 함수는 함수 객체를 할당한 변수를 미리 만들지만, var 키워드는 변수만 생성하고 값은 해당 행의 코드가 실행될 때 할당된다.
- 이미 변수가 생성되어 있다면 선언적 함수와 var 키워드로 선언된 변수는 다음 작업을 한다.
  - 선언적 함수 : 이미 변수가 생성되어 있으므로 기존 값을 새로운 함수로 변경한다.
  - var 키워드 : 이미 변수가 생성되어 있으므로 변수를 생성하지 않는다.

함수의 정의(3)

다음 코드의 실행 순서

- 오류가 발생할 것만 같지만…

[예제 03-02]

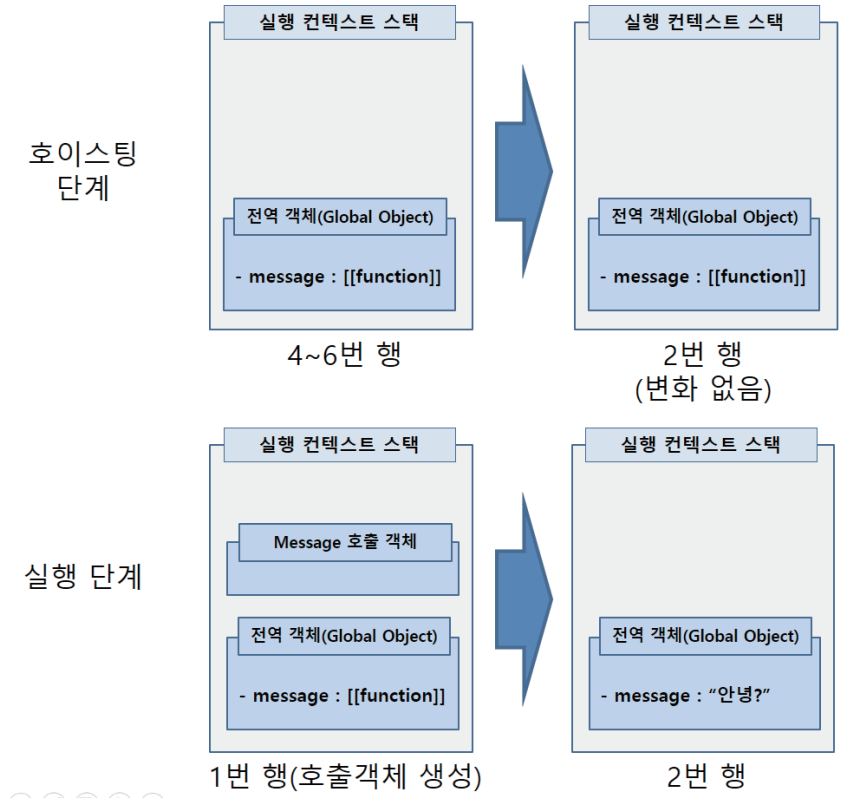
```
01: console.log(message("홍길동"));
02: var message = "안녕?";
03: console.log(message);
04: function message(name) {
05:     return "hello " + name;
06: }
```

호이스팅 단계

- 04~06행 : message 변수에 함수가 미리 할당됨.
- 02행 : message 변수를 생성하려 시도. 이미 변수가 존재하므로 생성하지 않음.

실행 단계

- 01행 : message 변수의 함수 기능을 실행하여 결과 출력
- 02행 : message 변수에 문자열 할당. 데이터 타입이 function에서 string으로 변경됨.
- 03행 : message 변수값 출력



함수의 정의(4)

- 이건 함수는 아니지만..
  - 오류가 발생하지 않는다..
  - 호이스팅 단계에서는 변수를 위한 메모리만 할당하고 값은 할당하지 않는다.

[예제 03-03]

```
01: console.log(v1);
02: var v1 = 100;
03: console.log(v1);
04: var v1 = "hello";
05: console.log(v1);
```

실행 결과

```
undefined
100
hello
```

함수의 정의(5)

선언적 함수의 중복

- 이것도 오류가 발생하지 않는다.
- 직전 예제와는 달리 호이스팅 단계에서 값을 할당해버린다.

[예제 03-04]

```
01: function calc(x,y) {
02:     return x+y;
03: }
04: console.log(calc(5,3));
05:
06: function calc(x,y) {
07:     return x*y;
08: }
```

호이스팅 단계

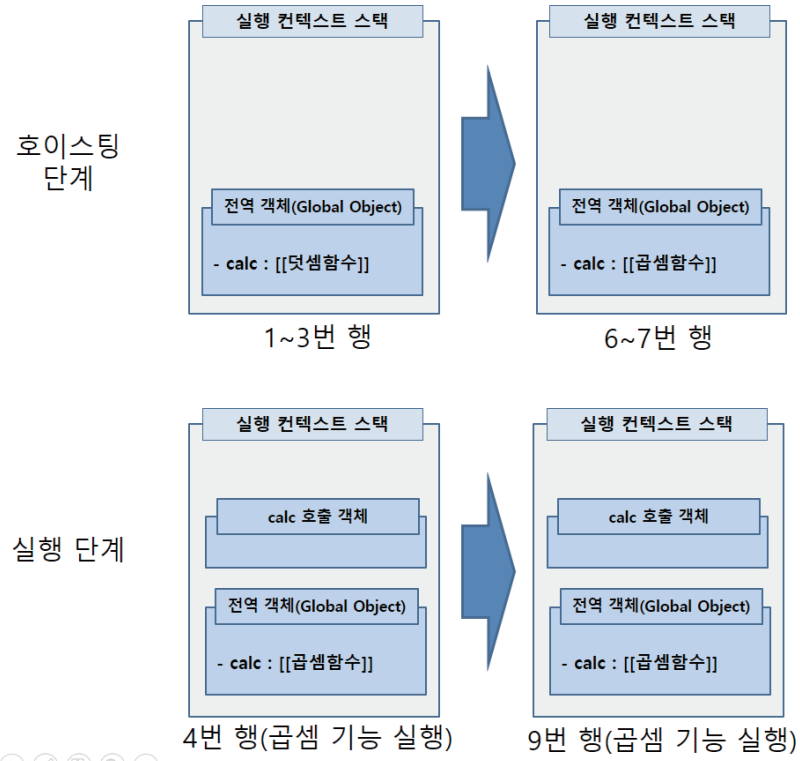
- 01~03행 : calc 변수를 생성하고 함수를 할당함.
- 06~08행 : 기존 calc 변수에 곱셈 기능을 가진 새로운 함수를 할당함.

실행 단계

04행 : 곱셈 기능 실행
09행 : 곱셈 기능 실행

실행 결과

```
15
15
```



함수의 정의(6)

함수리터럴 방식에서의 중복

- 역시 오류는 발생하지 않으나 var 키워드의 접근방식

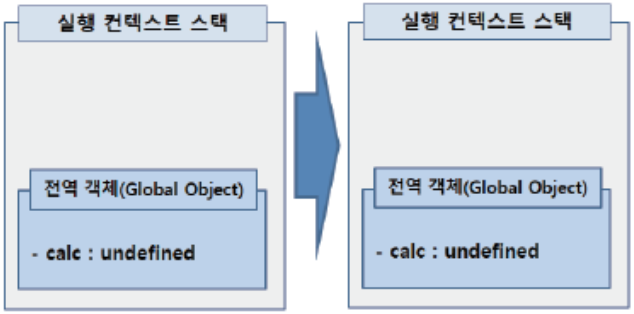
[예제 03-05]

```
01: var calc = function(x,y) {
02:     return x+y;
03: };
04: console.log(calc(5,3));
05:
06: var calc = function(x,y) {
07:     return x*y;
08: };
09: console.log(calc(5,3));
```

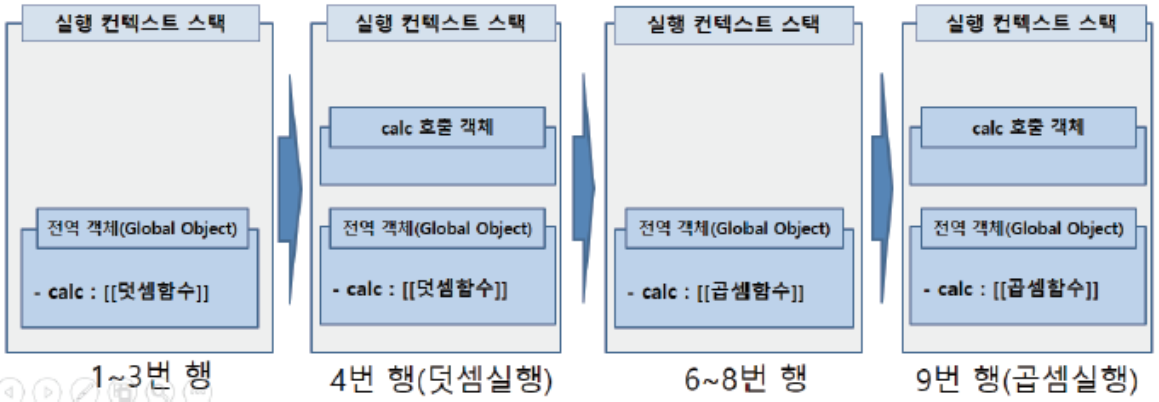
실행 결과

```
8
15
```

호이스팅 단계



실행 단계



함수의 정의(7)

- 함수 리터럴 방식에서의 함수명은 적용되지 않음

[예제 03-06]

```
var A1 = function A2() {
    console.log("hello");
}
A1();
A2();
```

실행 결과

```
hello
A2 is not defined
```

함수의 파라미터(1)

명시적 파라미터 사용

[예제 03-07]

```
function test(a,b,c) {
    console.log(a);
    console.log(b);
    console.log(c);
}

test(100,200,300);              //호출1 : 100, 200, 300
test(100);                      //호출2 : 100, undefined, undefined
test(100,200,300,400,500);      //호출3 : arguments 객체 확인
```

- 3번의 호출 모두 오류가 발생하지 않음
  - 2번째 : 명시적 파라미터 b,c가 전달되지 않았으므로 undefined
  - 3번째 : 명시적 파라미터 a,b,c에는 100,200,300이 전달되었지만? 나머지는?
  - arguments를 확인해보면 알 수 있다.

함수의 파라미터(2)

arguments

- 함수 호출시에 전달된 파라미터들은 유사배열인 arguments에 전달
- 인덱스번호를 이용해 접근 가능
- callee는 현재 호출 중인 함수를 가리키는 내부 참조이다

```
function test(a,b,c) {
    console.log(a);
    console.log(b);
    console.log(c);
    console.dir(arguments);
}
```

```
100
200
300
▼Arguments[5] i
   0: 100
   1: 200
   2: 300
   3: 400
   4: 500
  ▼callee: function test(a,b,c)
     arguments: null
     caller: null
     length: 3
     name: "test"
    ▶prototype: test
    ▶__proto__: function ()
    ▶<function scope>
   length: 5
  ▶Symbol(Symbol.iterator): function values()
  ▶__proto__: Object
```

함수의 파라미터(3)

- 아래 두 코드는 결과적으로 동일한 기능 수행

```
function test(a,b) {

}
```

```
function test() {
    var a = arguments[0];
    var b = arguments[1];
}
```

- 함수의 파라미터로 다른 함수를 전달할 수 있음

[예제 03-08]

```
01: function test(callback) {
02:     if (typeof callback != "function") {
03:         throw "callback 파라미터값으로 함수만 전달 가능합니다";
04:     }
05:     callback();
06: }
07:
08: test(function() {
09:     console.log("hello");
10: })
```

실행 결과

```
hello
```

함수의 리턴값(1)

함수는 값을 리턴할 수 있음.

- 값을 리턴하지 않는 함수로부터 값을 리턴받으려 하면?
  - 값이 없으니 undefined

[예제 03-09]

```
01: function add(x,y) {
02:     return x+y;
03: }
04:
05: function test() {
06:     //return문이 없는 함수
07: }
08:
09: var result1 = add(4,5);
10: console.log(result1);
11:
12: var result2 = test();
13: console.log(result2);
```

실행 결과

```
9
undefinded
```

함수의 리턴값(2)

함수는 함수를 리턴할 수 있음.

- 함수는 일급객체이므로 당연히 가능함.

[예제 03-10]

```
01: var setmessage = function(message) {
02:     return function(name) {
03:         return message + " " + name;
04:     }
05: }
06:
07: var m = setmessage("hello");
08: console.log(m("홍길동"));
09: console.log(m("이몽룡"));
10: console.log(m("성춘향"));
```

실행 결과

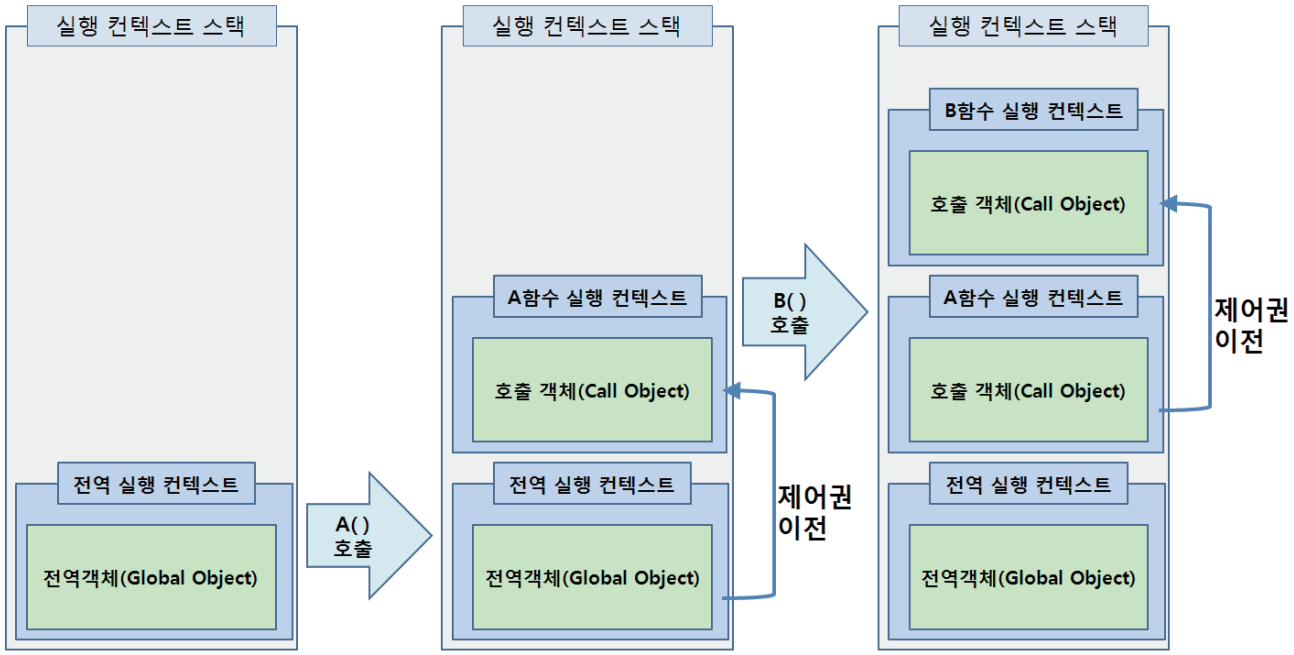
```
hello 홍길동
hello 이몽룡
hello 성춘향
```

함수의 호출과정(1)

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

[예제 03-11]

```
01: function A() {
02:     B();
03:     console.log("world");
04: }
05: function B() {
06:     console.log("hello");
07: }
08: A();
```



함수의 호출과정(2)

함수 호출 과정에서의 호이스팅

가) 함수 실행 컨텍스트를 생성하여 실행 컨텍스트 스택(Execution Context Stack)에 추가한다.

나) 실행 컨텍스트 안에 호출 객체(Call Object)를 만든다.

다) 호출 객체 안에 함수 호출 시에 전달되는 파라미터값과 arguments 객체를 생성한다.

라) 스코프 체인(scope chain) 정보를 생성한다.

마) 호출 객체 안에 선언적 방식의 내부 함수를 미리 만든다(함수까지 미리 할당).

바) 호출 객체 안에 var 키워드로 선언된 변수를 미리 만든다(변수만 만들고 값 할당은 하지 않음).

사) this값을 바인딩한다.

다) 함수 내부의 코드를 실행한다.

- 스코프 체인은 리스트 형태의 구조
  - 현재 호출 중인 함수가 정의된 호출 객체, 전역 객체를 가리키는 정보를 가지고 있음

함수의 호출과정(3)

- 다음 예제를 통해 실행 과정을 살펴보자.

[예제 03-12]

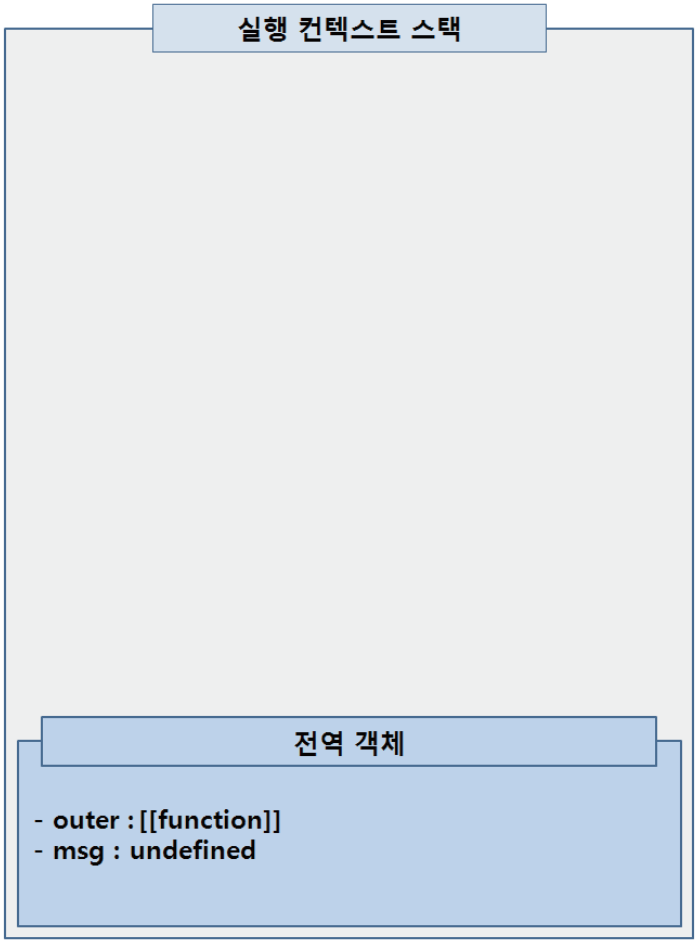
```
01: var msg = "GLOBAL";
02: function outer() {
03:     var msg = "OUTER";
04:     console.log(msg);
05:     inner();
06:     function inner() {
07:         var msg = "INNER";
08:         console.log(msg);
09:     }
10: }
11: outer();
```

실행 결과

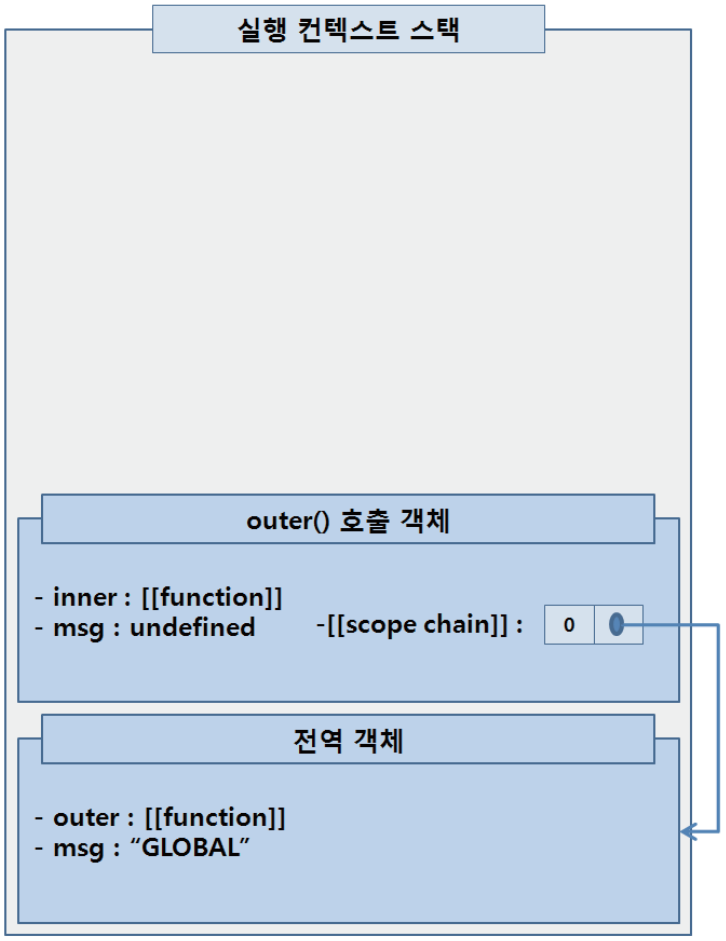
```
OUTER
INNER
```

함수의 호출과정(4)

- 1단계

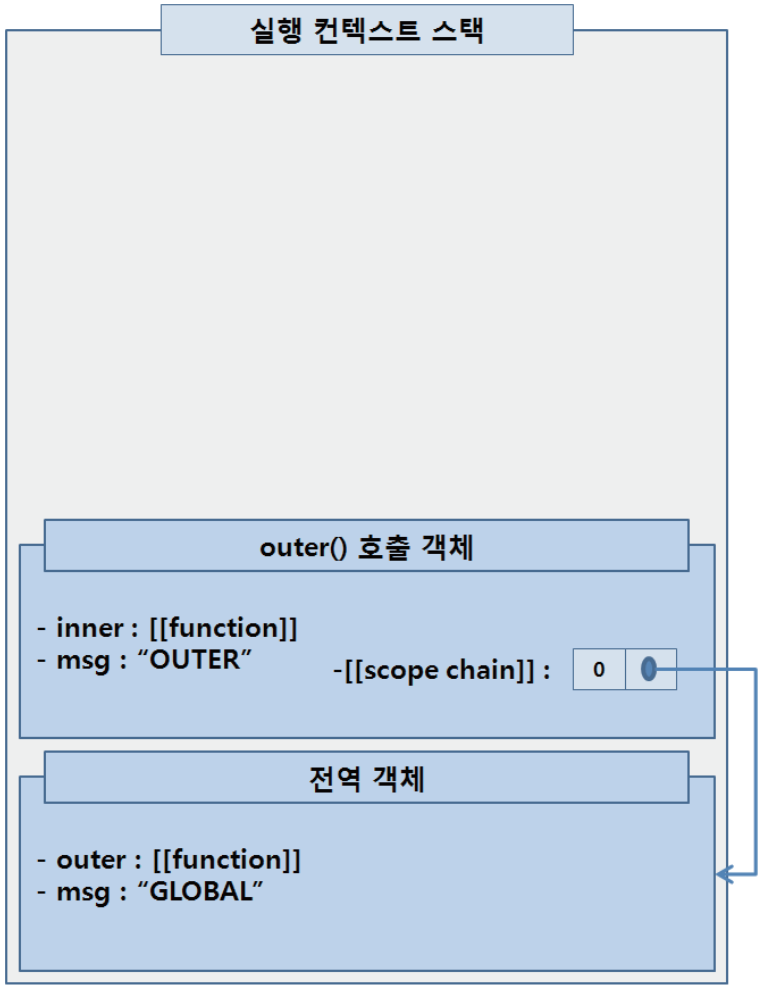


- 2단계

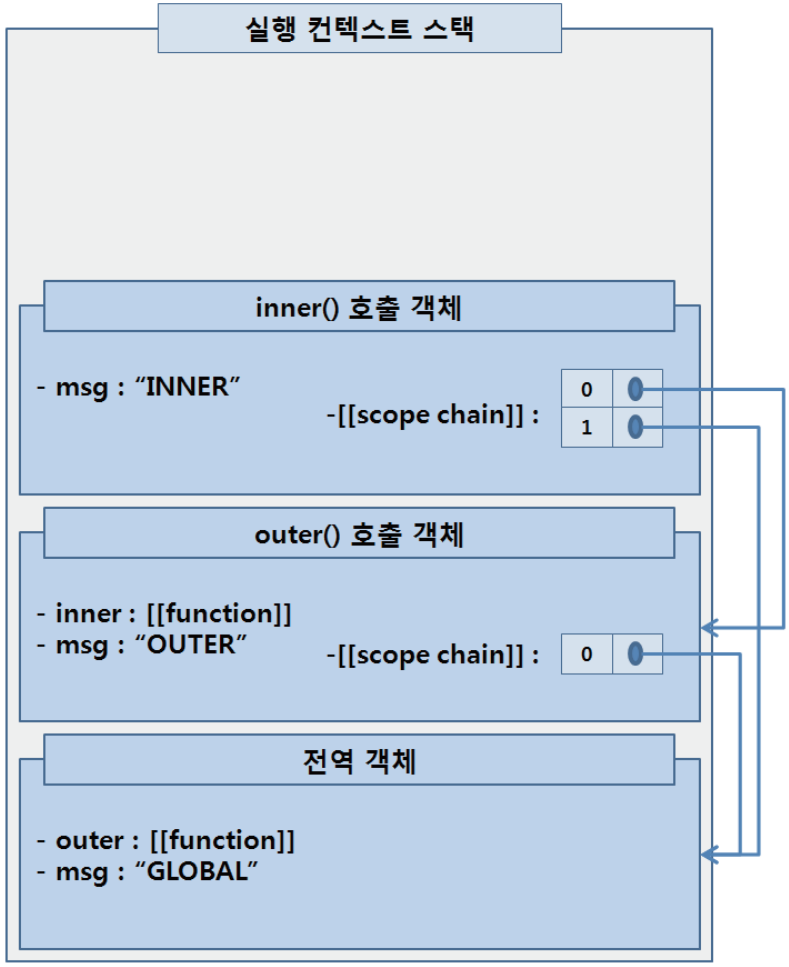


함수의 호출과정(5)

- 3단계



- 4단계



함수의 호출과정(6)

- 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

스코프와 스코프 체인(1)

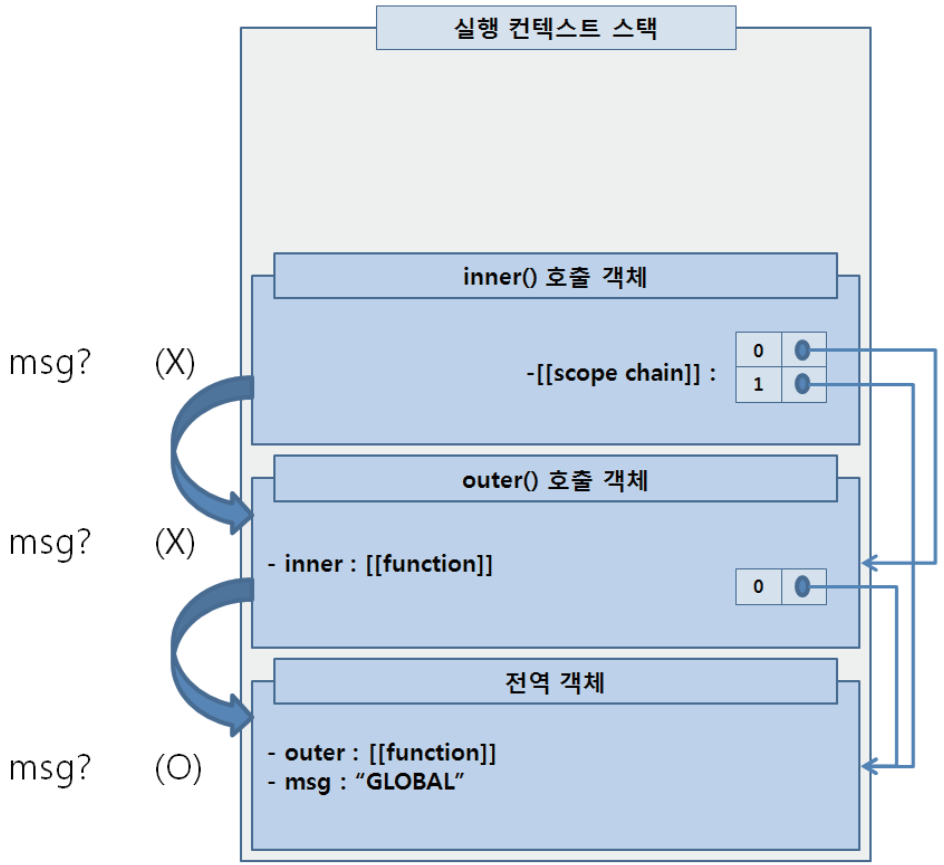
- 호출 객체 단위로 스코프가 결정된다
- 호출 객체 내에 스코프 체인에 대한 정보를 가지고 있다

[예제 03-13 : 3행과 7행 주석 처리]

```
01: var msg = "GLOBAL";
02: function outer() {
03:     //var msg = "OUTER";
04:     console.log(msg);
05:     inner();
06:     function inner() {
07:         //var msg = "INNER";
08:         console.log(msg);
09:     }
10: }
11: outer();
```

실행 결과

```
GLOBAL
GLOBAL
```



스코프와 스코프 체인(2)

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

[예제 03-14]

```
01: function test() {
02:     var num = 100;
03:     for (var num=0; num < 10; num++) {
04:         console.log(num);
05:     }
06:     return num;
07: }
08:
09: var result = test();
10: console.log("최종 결과 : " + result);
```

실행 결과

```
0
1
2
3
4
5
6
7
8
9
최종 결과 : 10
```

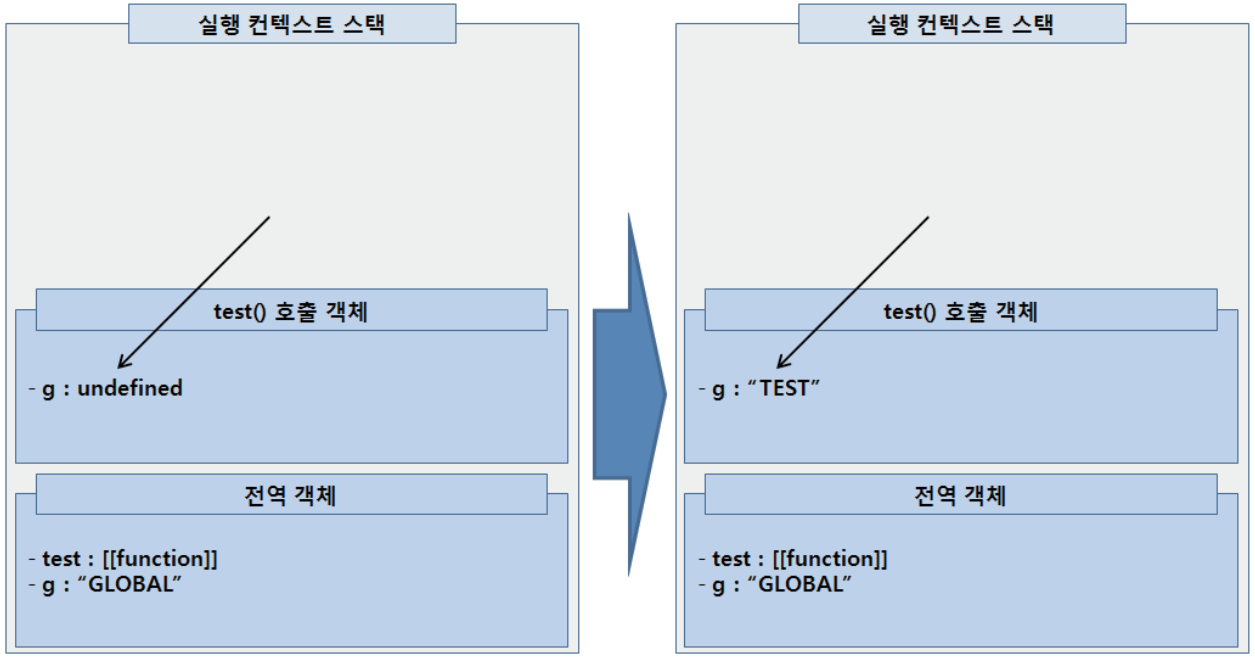
스코프와 스코프 체인(3)

함수 내부의 var 변수는 함수 전체에서 이용 가능

- 4행의 g는 호이스팅될 때 만들어지므로 3행에서 접근하는 g는 undefined 인 지역변수 g이다.

[예제 03-15]

```
01: var g = "GLOBAL";
02: function test() {
03:     console.log(g);
04:     var g = "TEST";
05:     console.log(g);
06: }
07: test();
```



3번행이 실행될 때

5번행이 실행될 때

스코프와 스코프 체인(4)

- 심지어는 다음 코드의 5행은 실행될 수 없는 코드이지만
  - 호이스팅 단계에서 지역 변수 g를 미리 만든다.

[예제 03-16]

```
01: var g = "GLOBAL";
02: function test() {
03:     console.log(g);
04:     if (false) {
05:         var g = "TEST";
06:     }
07:     console.log(g);
08: }
09: test();
```

  - 결과는 undefined, undefined

스코프 연습 문제(1)

첫번째

```
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"

스코프 연습 문제(2)

두번째

- 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 1이 그대로 유지되었음)

스코프 연습 문제(3)

세번째

```
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( ) { })( );

클로저(1)

클로저의 정의

"외부 함수 내의 내부 함수가 전역에서 참조되고, 내부 함수가 외부 함수 내의 지역 변수를 이용할 수 있게 되는 현상 또는 그 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 것"

"특정 함수 내의 지역 변수를 외부에서 접근할 수 있도록 하는 내부 함수"

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

클로저(2)

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

[예제 03-18]

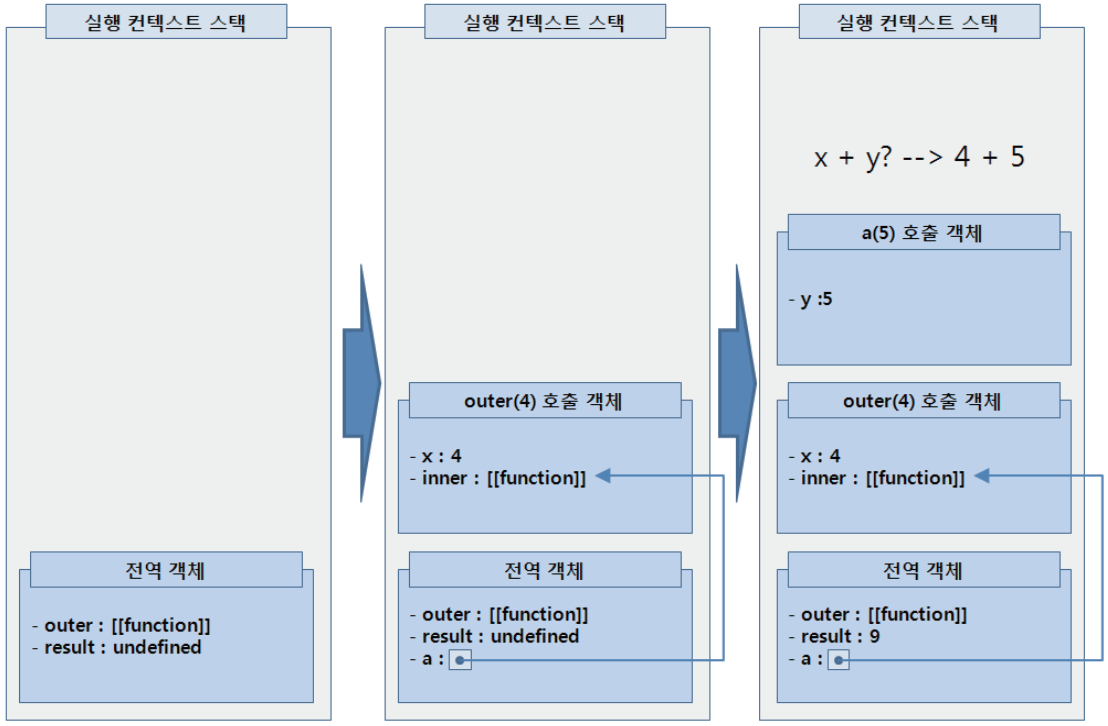
```
01: function outer(x) {
02:     function inner(y) {
03:         return x+y;
04:     }
05:     return inner;
06: }
07: window.a = outer(4);
08: var result = a(5);
09: console.log(result); //실행 결과 9
```

[예제 03-19]

```
01: function outer(g, x) {
02:     function inner(y) {
03:         return x+y;
04:     }
05:     g.a = inner;
06: }
07: outer(window, 4);
08: var result = a(5);
09: console.log(result); // 실행 결과 9
```

클로저(3)

- (이전 페이지에 이어서)




```
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

- 클로저 함수 내부의 자유변수는 오로지 클로저 함수를 통해서만 접근이 가능함.

클로저(4)

클로저가 여러개 생성되는 경우

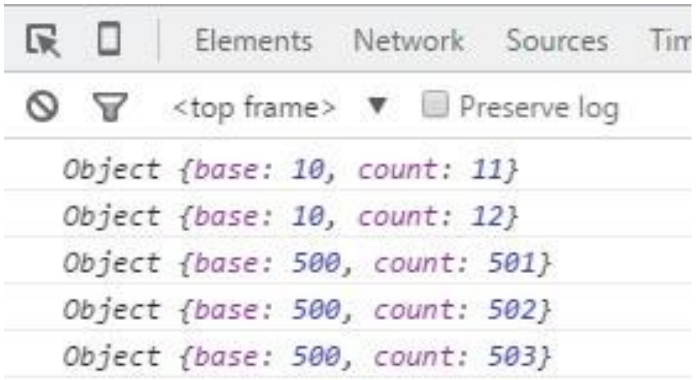
- 단지 함수의 중첩구조만 고려해서는 안됨

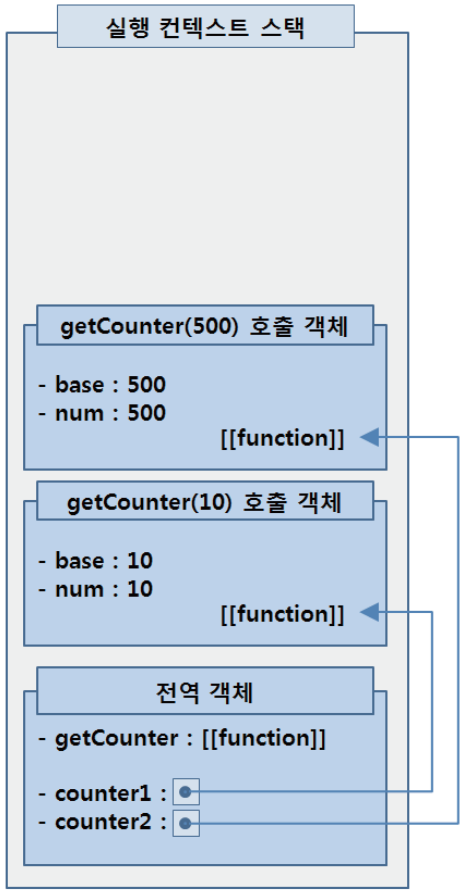
[예제 03-20]

```
01: function getCounter(base) {
02:     var num = base;
03:     return function() {
04:         return { base: base, count: ++num };
05:     }
06: }
07:
08: var counter1 = getCounter(10);
09: var counter2 = getCounter(500);
10: console.log(counter1());
11: console.log(counter1());
12: console.log(counter2());
13: console.log(counter2());
14: console.log(counter2());
```

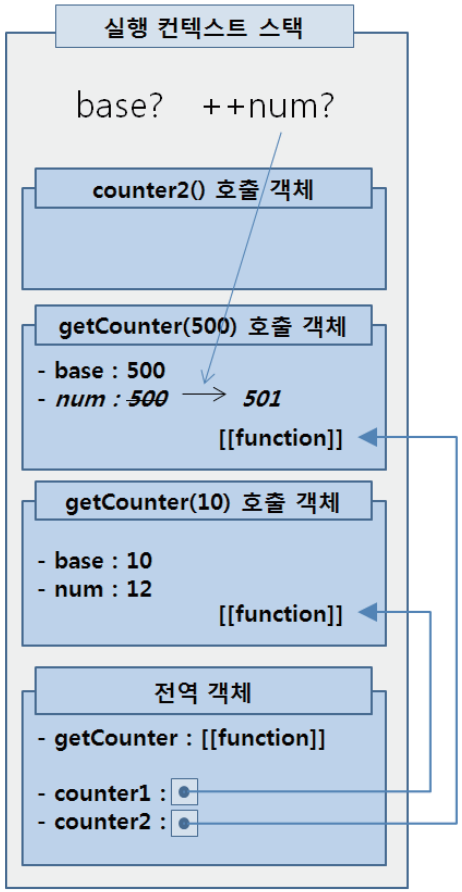
클로저(5)

- (이전 페이지에 이어서)
  - A – 9행까지의 실행
  - B – 9행 이후의 실행





<A>



<B>

클로저(6)

클로저의 사용 용도

- 캡슐화!!

[예제03-21 : nocap.js]

```
01: var calc = function(x, y) {
02:     var result = x/y;
03:     return util(result);
04: };
05: //반올림 처리 함수
06: var util = function(num) {
07:     return Math.round(num);
08: }
```

- 두 util 함수는 충돌남

[예제03-22]

```
01: <!DOCTYPE html>
02: <html lang="">
03: <head>
04: <meta charset="utf-8">
05: <title></title>
06: <script type="text/javascript" src="nocap.js"></script>
07: <script type="text/javascript">
08: function util() {
09:     return "hello jQuery!!";
10: }
11: </script>
12: <script type="text/javascript">
13: var divresult = calc(100, 6);
14: console.log(divresult);
15: </script>
16: </head>
17: <body>
18: </body>
19: </html>
```

클로저(7)

- 충돌 방지를 위해 즉시실행함수로 캡슐화

[예제 03-23 : cap.js]

```
01: (function(window) {
02:     var calc = function(x, y) {
03:         var result = x/y;
04:         return util(result);        //반올림
05:     };
06:
07:     var util = function(num) {
08:         return Math.round(num);
09:     }
10:
11:     window.calc = calc;
12: })(window);
```

[예제 03-24 : jQuery 2.1 골격]

```
01: (function( window, undefined ) {
02:    var
03:       rootjQuery,
04:       readyList,
05:       ...
06:       jQuery = function( selector, context ) {
07:          return new jQuery.fn.init( selector, context, rootjQuery );
08:       };
09:       ...
10:       completed = function() {
11:          ...
12:       };
13:
14:    jQuery.fn = jQuery.prototype = {
15:       jquery: core_version,
16:       constructor: jQuery,
17:       init: function( selector, context, rootjQuery ) {
18:          var match, elem;
19:          ...
20:          return jQuery.makeArray( selector, this );
21:       };
22:       ...
23:    }
24:    ...
25:
26:    window.jQuery = window.$ = jQuery;
27:    ...
28: })(window);
```

클로저 사용시 주의사항(1)

메모리 낭비와 성능 문제

- 클로저는 호출 객체를 만든 후 가비지 컬렉션하지 않고 유지하기 때문에 스크립트의 실행 속도를 느리게 하고 메모리 사용량도 증가시킨다.
  - 따라서 남용하는 것은 바람직하지 않음

[예제 03-25]

```
01: function Person(name, age) {
02:     this.name = name.toString();
03:     this.age = parseInt(age);
04:
05:     this.getInfo = function() {
06:         return this.name + "님의 나이 : "+this.age;
07:     }
08: }
09:
10: var p1 = new Person("홍길동", 20);
11: console.log(p1);
```

클로저 사용시 주의사항(2)

반복문에서의 클로저

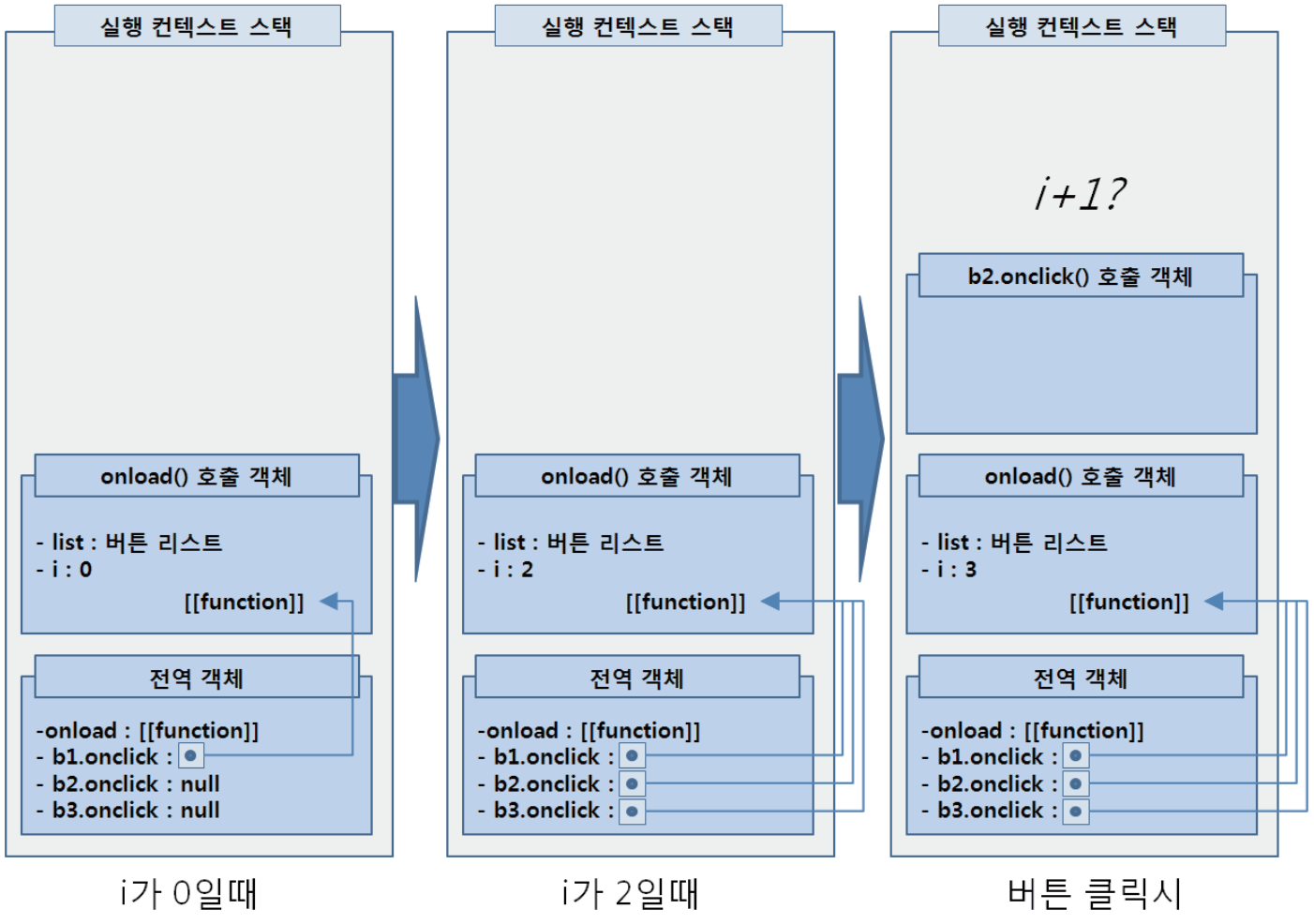
- 반복문에서 무심히 코드를 작성하게 되면 클로저 상황이 발생할 수 있고, 이로 인해 오류가 발생하기도 한다.

[예제 03-27 : 반복문에서의 주의사항]

```
01: <!DOCTYPE html>
02: <html lang="">
03: <head>
04: <meta charset="utf-8">
05: <title></title>
06: <script type="text/javascript">
07: window.onload = function() {
08:     var list =  document.getElementsByTagName("input");
09:     for (var i=0; i < list.length; i++) {
10:         list[i].onclick = function() {
11:             alert((i+1) + "번째 버튼 클릭!");
12:         }
13:     }
14: }
15: </script>
16: </head>
17: <body>
18:     <input id="b1" class="test" type="button" value="1번째 버튼" />
19:     <input id="b2" class="test" type="button" value="2번째 버튼" />
20:     <input id="b3" class="test" type="button" value="3번째 버튼" />
21: </body>
22: </html>
```

클로저 사용시 주의사항(3)

- (이어서)



클로저 사용시 주의사항(3)

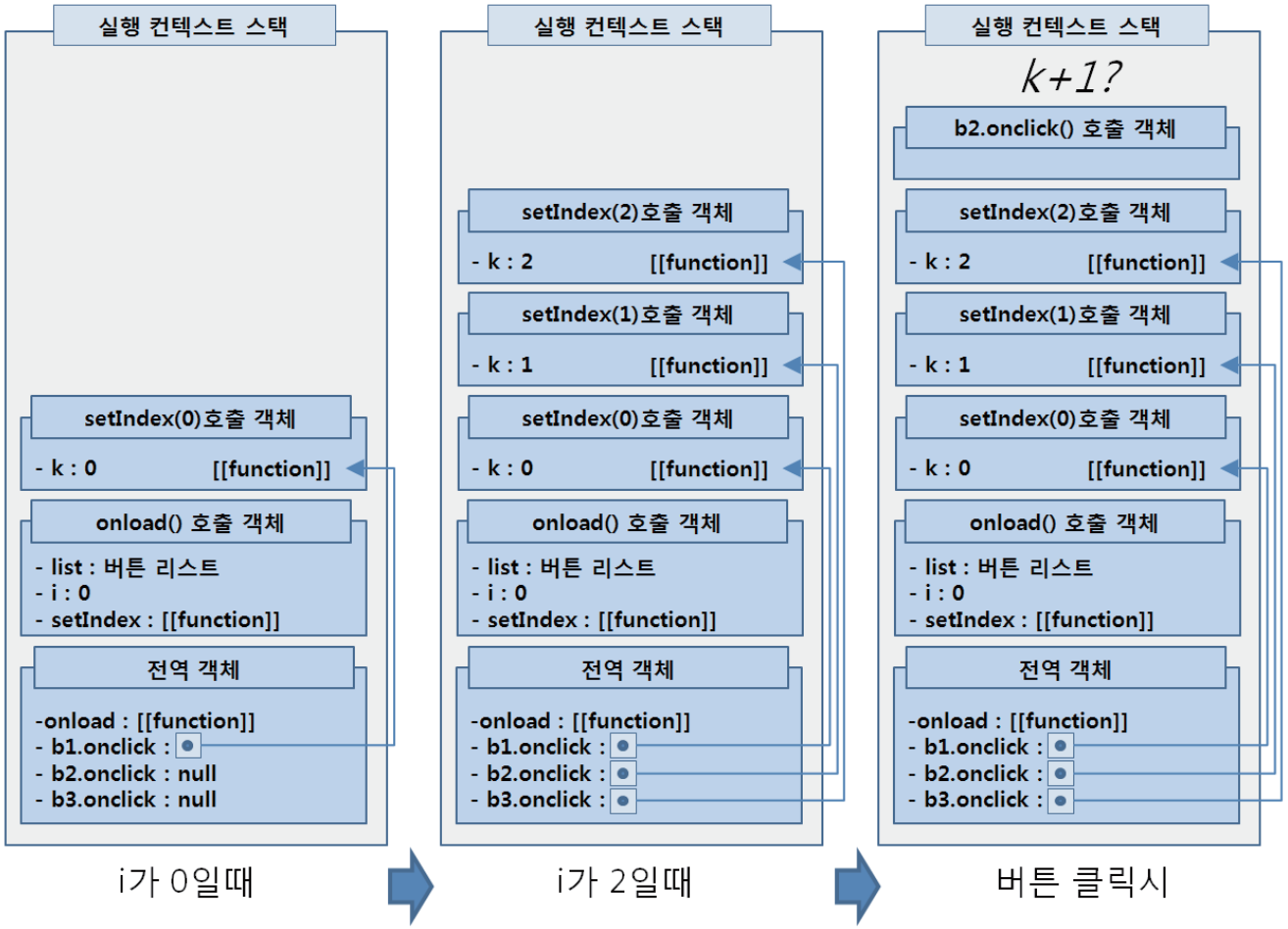
- (이어서)
  - 문제해결!! 반복문에서 함수 호출 --> 클로저 생성

```
06: <script type="text/javascript">
07: window.onload = function() {
08:     var list =  document.getElementsByTagName("input");
09:     for (var i=0; i < list.length; i++) {
10:         setIndex(i);
11:     }
12:
13:     function setIndex(k) {
14:         list[k].onclick = function() {
15:             alert((k+1) + "번째 버튼 클릭!");
16:         }
17:     }
18: }
19: </script>
```

클로저 사용시 주의사항(4)

(이어서)

- 문제해결



클로저 사용시 주의사항(5)

this와 arguments

- 내부 함수에서 외부 함수의 this와 arguments는 참조할 수 없다
  - this와 arguments는 호출 객체가 생성될 때마다 자동으로 만들어지는 호출 객체의 속성이다.
  - 명시적 파라미터가 없다 하더라도 길이가 0인 arguments 객체는 생성되므로 내부 함수에서 외부 함수의 arguments, this에는 직접 접근할 수 없다.
- 외부 함수 안에서 this, arguments를 다른 변수에 할당해 저장해두고 내부 함수에서 접근하도록 코드를 작성해야 한다.

[예제 03-29]

```
01: function outer() {
02:     var outerargs = arguments;
03:     inner();
04:     function inner() {
05:         console.log(outerargs);
06:         console.log(arguments);
07:     }
08: }
09:
10: outer();
```

jquery와 클로저

- jQuery와 클로저
  - jQuery와 jQuery 플러그인 상당수는 클로저를 이용해 개발
  - jQuery로 코드를 작성하다보면 의외로 클로저를 많이 사용함

```
09: <script type="text/javascript" src="https://code.jquery.com/jquery-3.1.0.js"></script>
10: <script type="text/javascript">
11: $(document).ready(function() {
12:     var duration = 300;
13:
14:     $("#setup").on("click", function() {
15:         duration = parseInt($("#speed").val());
16:         console.log(duration);
17:     });
18:
19:     $("#goeffect").on("click", function() {
20:         console.log(duration);
21:         $("#view1").slideToggle(duration);
22:     });
23: });
24: </script>
```