리액트를 위한 Javascript 심화

1. 자바스크립트의 실행 컨텍스트

❖실행 컨텍스트(Execution Context)란?

- 실행할 코드에 제공하는 실행 환경 정보를 모아놓은 객체
- 자바스크립트 코드들이 실행되기 위한 환경, 컨테이너

❖실행 컨텍스트 스택(Execution Context Stack)

- 콜 스택(Call Stack)이라고도 부름
- 실행 컨텍스트를 저장하는 스택 형태의 자료 구조
- 자바스크립트 런타임 엔진은 스택의 가장 상위의 실행 컨텍스트를 이용해 코드를 실행함

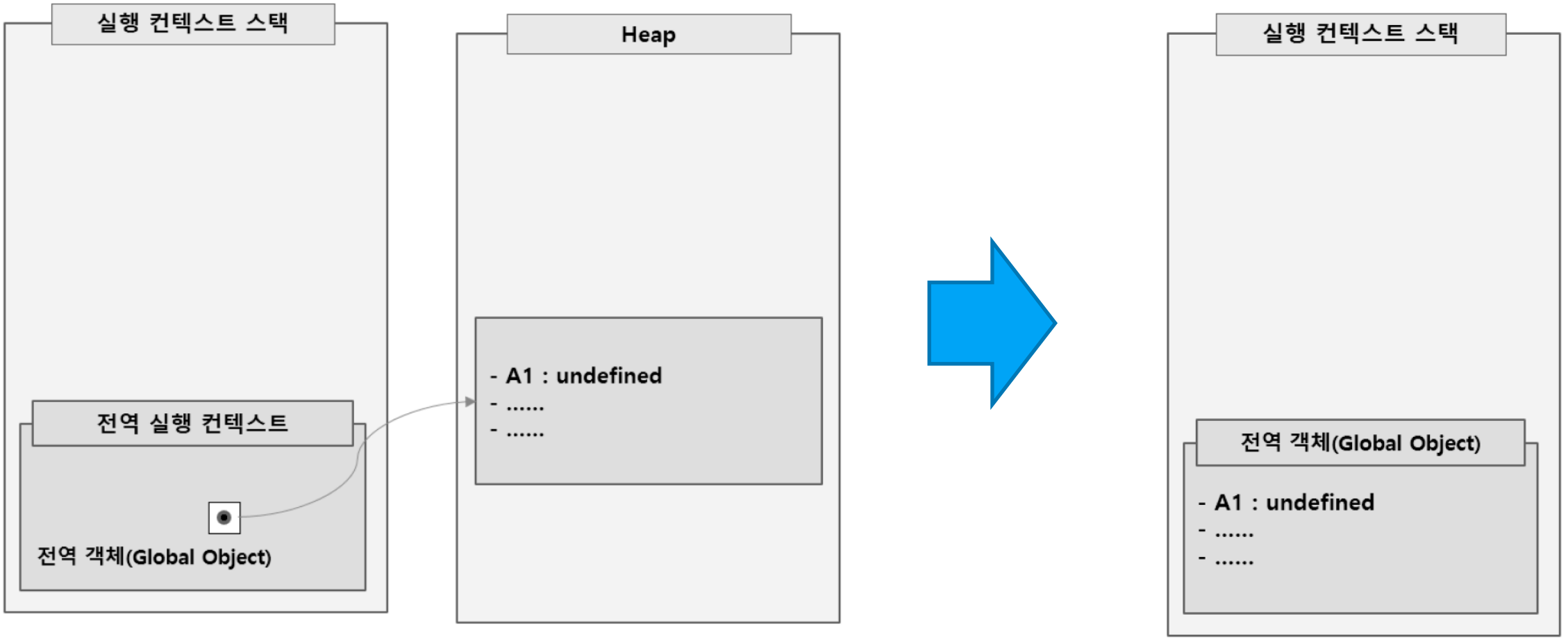
❖실행 컨텍스트 종류

- 전역 실행 컨텍스트(Global Execution Context)
  - 자바스크립트 코드가 처음 실행되는 순간 콜 스택에 가장 먼저 생성됨
  - 전역 실행 컨텍스트 내부에 전역 객체(Global Object) 생성
- 함수 실행 컨텍스트(Function Execution Context)
  - 함수가 호출되면 생성되어 콜스택에 쌓이는 실행 컨텍스트
  - 함수 실행 컨텍스트 내부에 호출 객체(Call Object) 생성
  - **함수 호출이 완료되면 스택에서 제거됨**

1. 자바스크립트의 실행 컨텍스트

❖전역 실행 컨텍스트와 전역 객체

- 편의상 오른쪽과 같이 표현함



2. 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. 함수

❖일급 객체로서의 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?

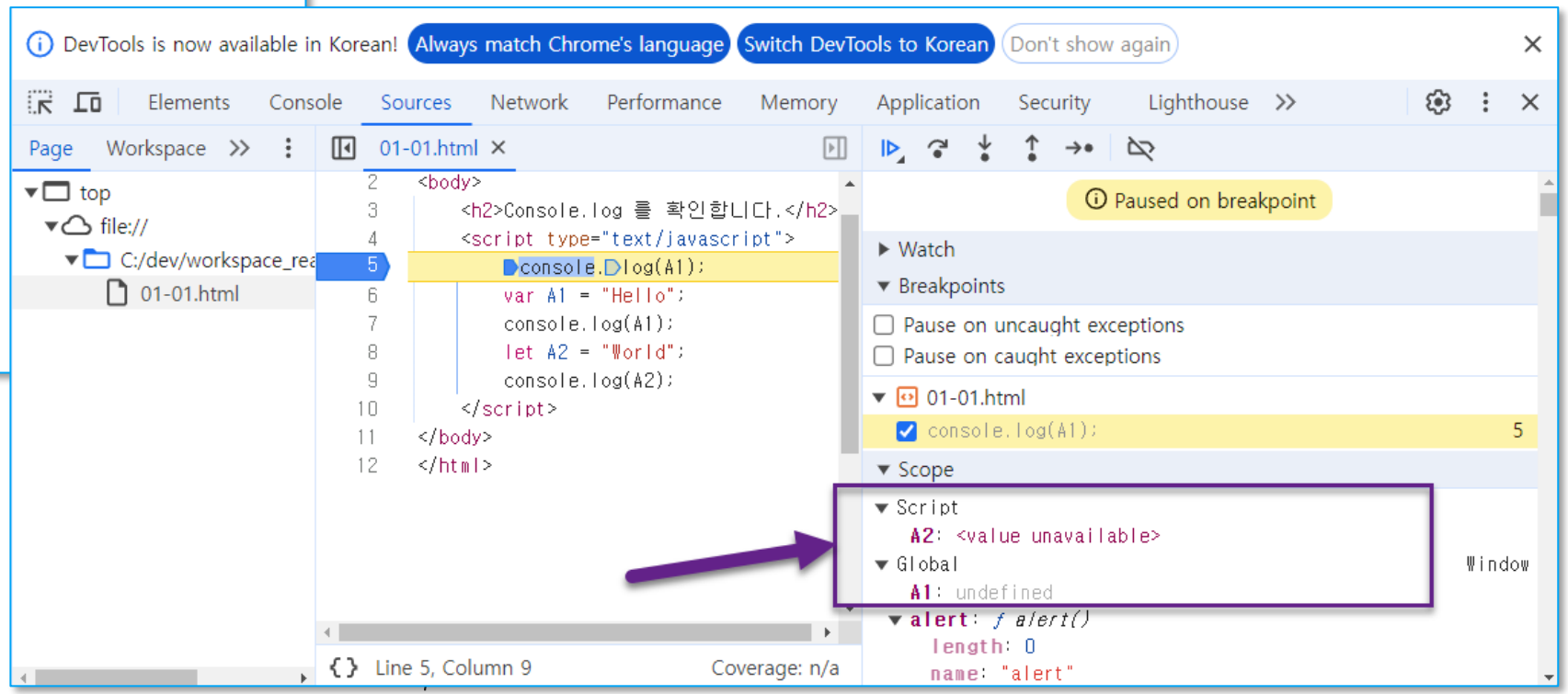
- 자바스크립트 런타임의 인터프리터가 변수와 함수를 위한 메모리 공간을 미리 할당하는 작업
  - 런타임은 자바스크립트 코드에서 변수와 함수를 미리 스캔하여 데이터 구조를 생성하고 메모리에 추가함
  - Hoisting --〉 Execution
- 호이스팅 대상
  - var 키워드로 선언한 변수
    - undefined로 초기화까지 완료됨
  - 선언적 방식의 함수
    - 함수가 초기화됨
- 호이스팅 대상이 아닌 것
  - let, const : 호이스팅 단계에서 스캔되지만 변수를 초기화하지 않음
  - 함수표현식으로 생성한 함수 : let, const와 같은 방식으로 처리함

3. 호이스팅(Hoisting)

❖ 다음 코드는 오류를 일으키지 않음. (예제 01-01)

- 호이스팅 단계에서 미리 A1 변수 선언! 브라우저 개발자 도구로 중단점 설정하여 확인

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        console.log(A1);
        var A1 = "Hello";
        console.log(A1);
        let A2 = "World";
        console.log(A2);
    </script>
</body>
</html>
```



3. 호이스팅(Hoisting)

❖호이스팅으로 인한 변수 중복 선언(예제 01-02)

- 호이스팅 단계에서 이미 변수가 만들어져 있다면 다시 생성하지 않음. 건너뜀(skip)
- 변수명이 중복되어도 오류가 발생하지 않으므로 주의해야 함.
- 리액트 앱을 개발할 때는 let, const를 사용하는 것이 바람직함.

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        var A1 = "hello";
        console.log(A1);
        var A1 = 1000;
        console.log(A1);
        var A1 = true;
        console.log(A1);
    </script>
</body>
</html>
```

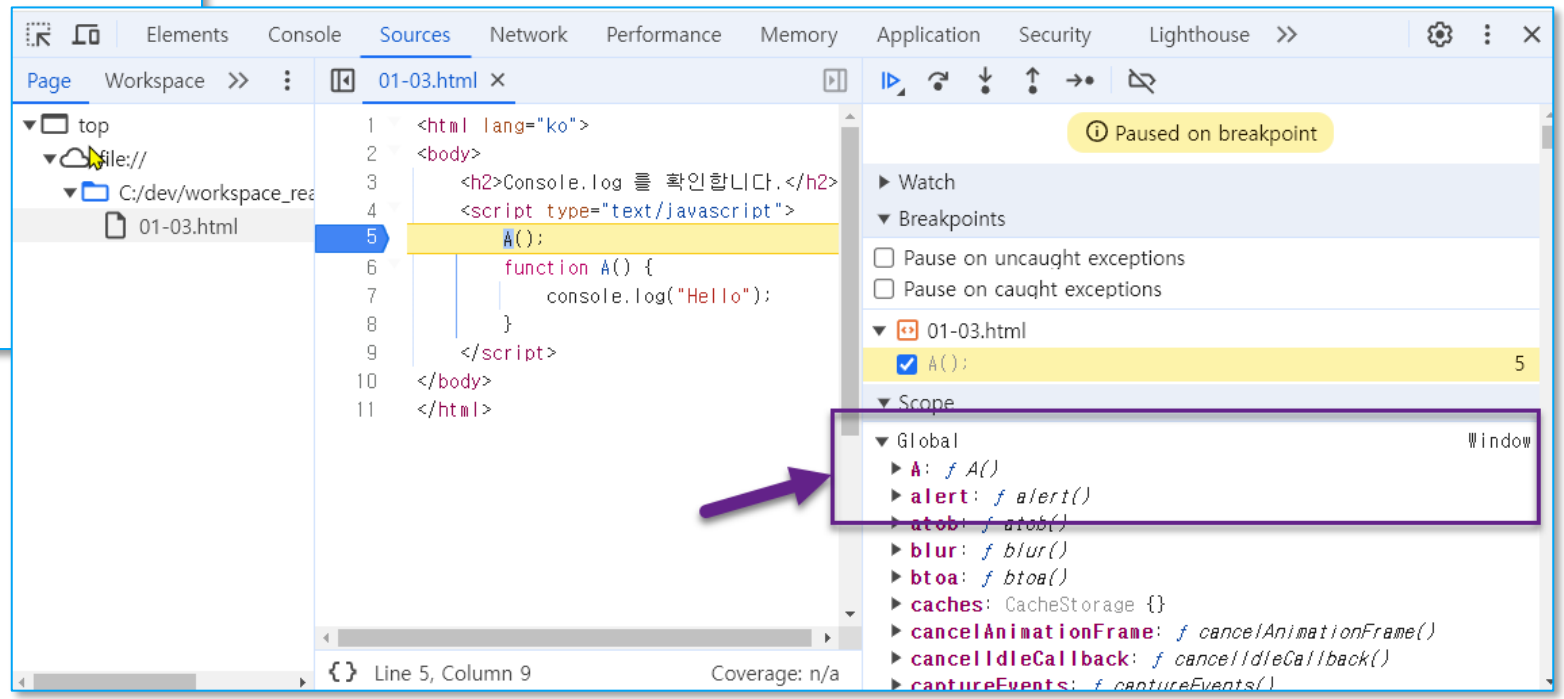
```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        var A1;

        A1 = "hello";
        console.log(A1);
        A1 = 1000;
        console.log(A1);
        A1 = true;
        console.log(A1);
    </script>
</body>
</html>
```

3. 호이스팅(Hoisting)

❖함수 호이스팅

- 선언적 함수는 호이스팅됨. 다음 코드는 오류 발생되지 않음(예제 01-03)

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        A();
        function A() {
            console.log("Hello");
        }
    </script>
</body>
</html>
```



3. 호이스팅(Hoisting)

❖선언적 함수의 중복 선언

- 다음 코드의 실행 결과는? (예제 01-04)
  - 에러 발생되지 않고 world 출력
- 호이스팅 과정에서 이미 선언된 것이 있다면 다시 선언함

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        function A() {
            console.log("Hello");
        }
        A();
        function A() {
            console.log("World");
        }
    </script>
</body>
</html>
```

3. 호이스팅(Hoisting)

❖선언적 함수와 변수의 중복 선언

- 자바스크립트에서는 함수도 변수라는 생각을 갖도록 하자.
- 다음 코드의 실행(예제 01-05)
  - world가 아닌 A 함수 코드가 출력

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        console.log(A);
        function A() {
            console.log("hello");
        }
        var A = "world";
    </script>
</body>
</html>
```

- 중복선언시 기억할 것
  - var는 건너뛰고 선언적 함수는 덮어씀

3. 호이스팅(Hoisting)

❖let, const, 함수 표현식

- 스캔은 되지만 호이스팅하지 않음.
  - 다음 코드들은 모두 오류 발생 (예제 01-06~07)
- 권장되는 방법

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        console.log(A1);
        let A1 = "hello";
    </script>
</body>
</html>
```

```
▶Uncaught ReferenceError: Cannot access 'A1' before initialization
    at 01-06.html:5:21
```

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        const A = () => {
            console.log("hello");
        }
        A();
        const A = () => {
            console.log("world");
        }
    </script>
</body>
</html>
```

```
Uncaught SyntaxError: Identifier 'A' has already been declared (at 01-07.html:9:15)
```

4. 함수 호출과 실행 컨텍스트

❖ 함수 호출 과정

- 함수 와 함수 내부 코드를 스캔한다.
- 함수 실행 컨텍스트(Function Execution Context) 를 생성하여 실행 컨텍스트 스택(콜스택)의 최상위에 추가한다.
- 함수 실행 컨텍스트 내부에 호출 객체(Call Object)를 생성한다.
- 호출 객체 안에 함수 호출시에 전달되는 파라미터 값과 arguments 객체를 생성한다.
- 범위 체인(Scope Chain) 정보를 생성한다.
- 함수 내부 코드에서 선언적 함수를 찾아 호출 객체 내부에 생성한다.
- 함수 내부 코드에서 var 키워드로 선언된 변수를 찾아 미리 생성한다.
- this를 연결한다.
- ----------여기까지가 호이스팅!!---------
- 함수 내부의 코드를 실행함
- 함수 호출이 완료되고나면 콜 스택에서 함수 실행 컨텍스트를 제거한다.

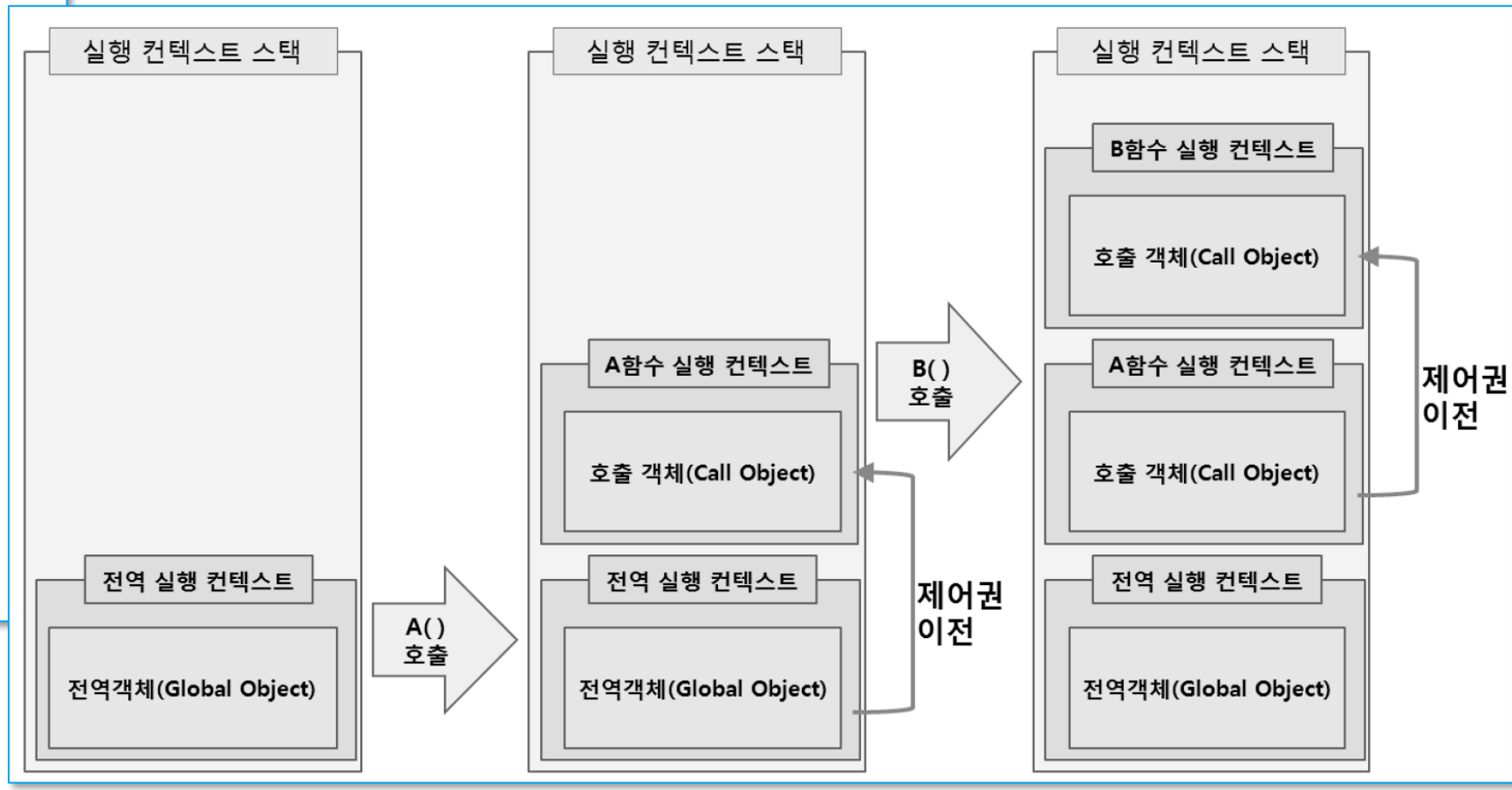
❖ 스코프 체인은 리스트 형태의 구조

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

4. 함수 호출과 실행 컨텍스트

❖기본 흐름 이해 (예제01-08)

```
<html lang="ko">
<body>
<h2>Console.log 를 확인합니다.</h2>
<script type="text/javascript">
const A = () => {
    console.log("hello");
}
const B = () => {
    A();
    console.log("world");
}
B();
</script>
</body>
</html>
```



4. 함수 호출과 실행 컨텍스트

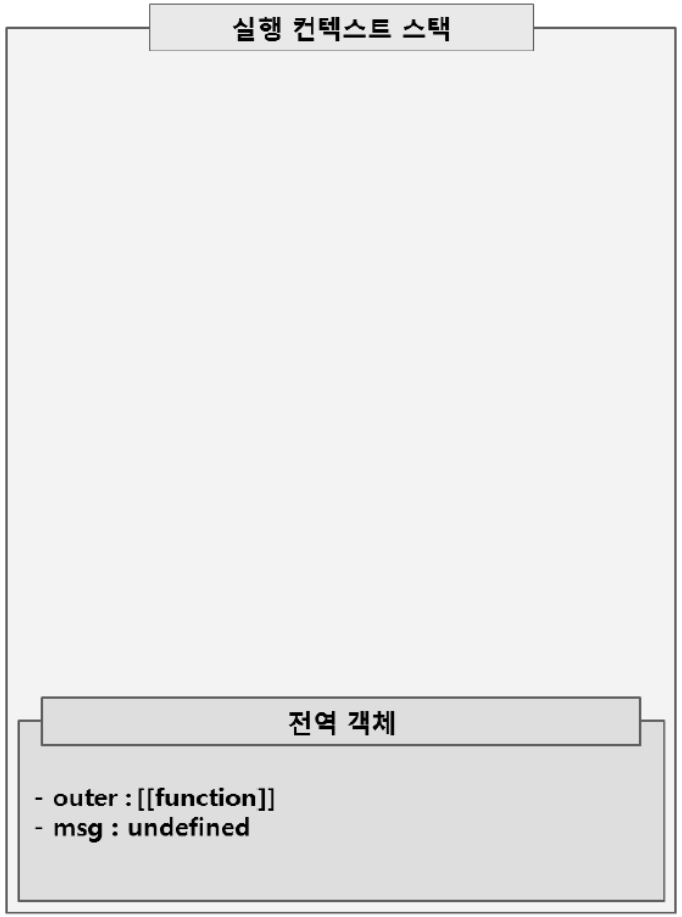
❖중첩된 함수 호출 과정 예(예제 01-09)

```
<html lang="ko">
<body>
    <h2>Console.log 를 확인합니다.</h2>
    <script type="text/javascript">
        var msg = "GLOBAL";
        function outer() {
            var msg = "OUTER";
            console.log(msg);
            inner();
            function inner() {
                var msg = "INNER";
                console.log(msg);
            }
        }
        outer();
    </script>
</body>
</html>
```

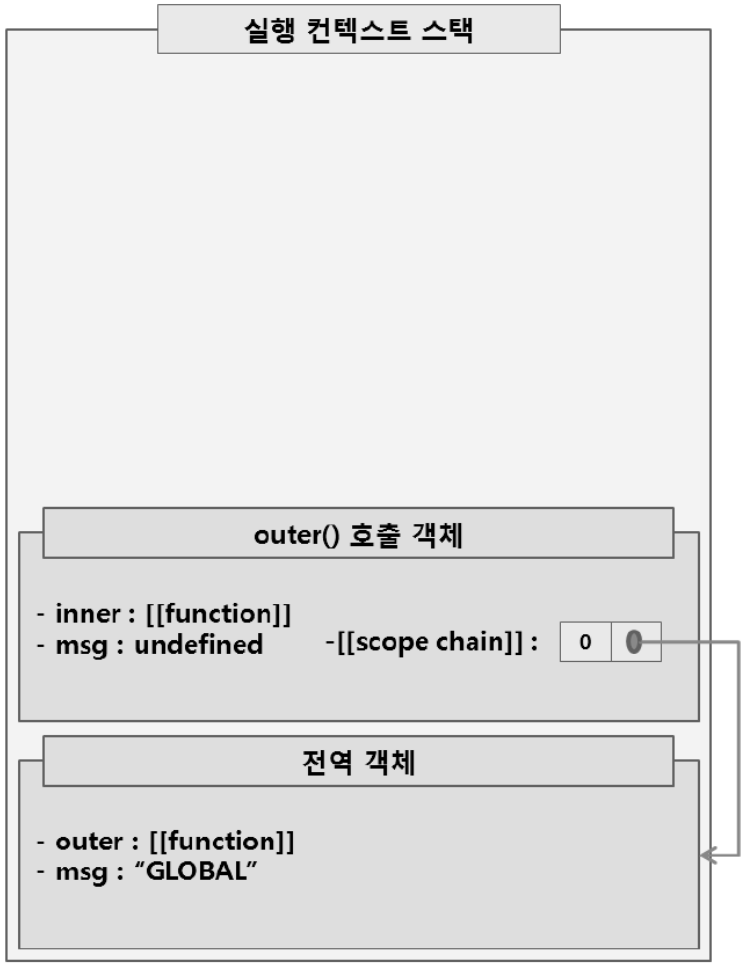
4. 함수 호출과 실행 컨텍스트

❖중첩된 함수 호출 과정 예 (이어서)

- 1단계



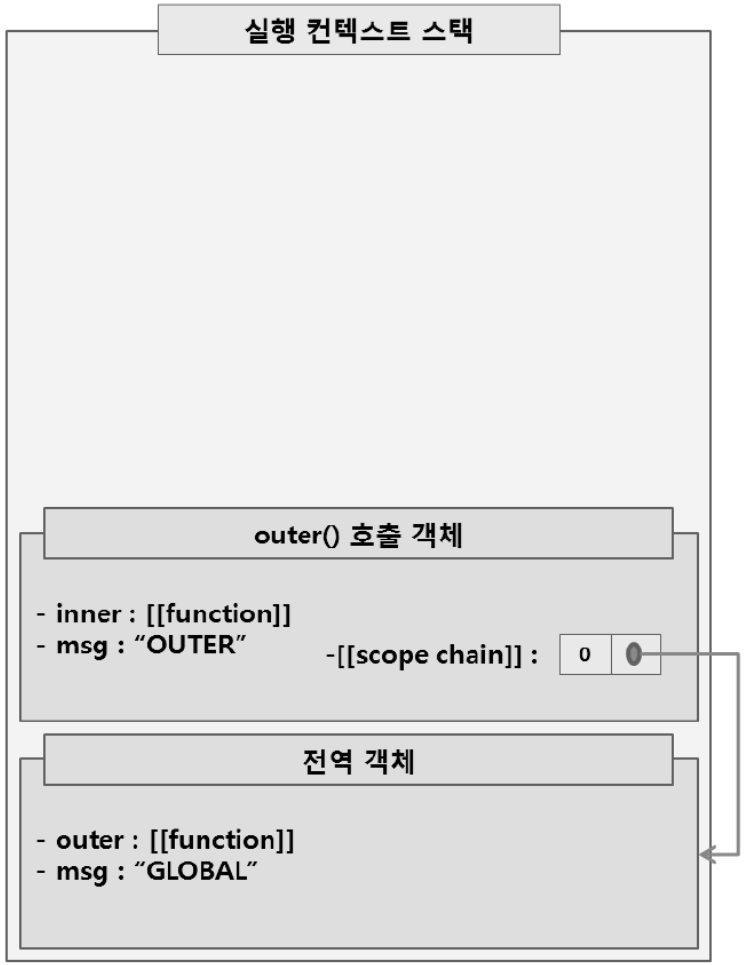
- 2단계



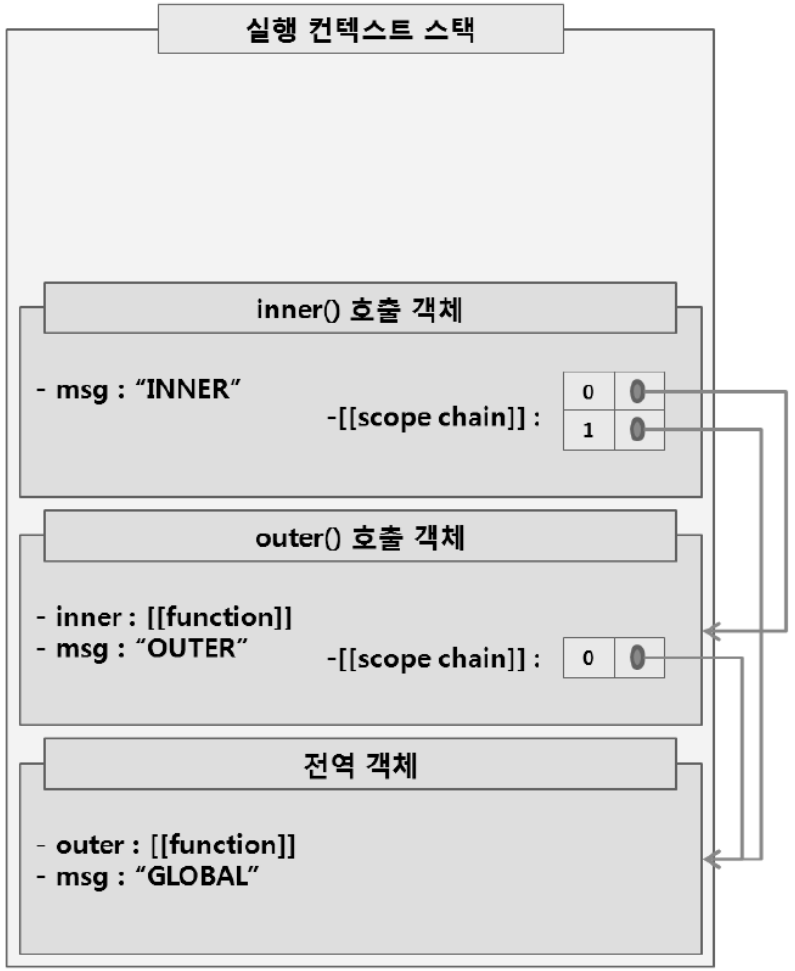
4. 함수 호출과 실행 컨텍스트

❖중첩된 함수 호출 과정 예 (이어서)

- 3단계



- 4단계



4. 함수 호출과 실행 컨텍스트

- 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

❖스코프(Scope)란?

- 변수에 접근할 수 있는 범위
- 호출 객체 단위로 스코프가 결정됨

❖스코프 체인

- 호출 객체가 스코프 체인에 대한 정보를 가지고 있음
- 참조 가능한 호출 객체에 대한 리스트 정보

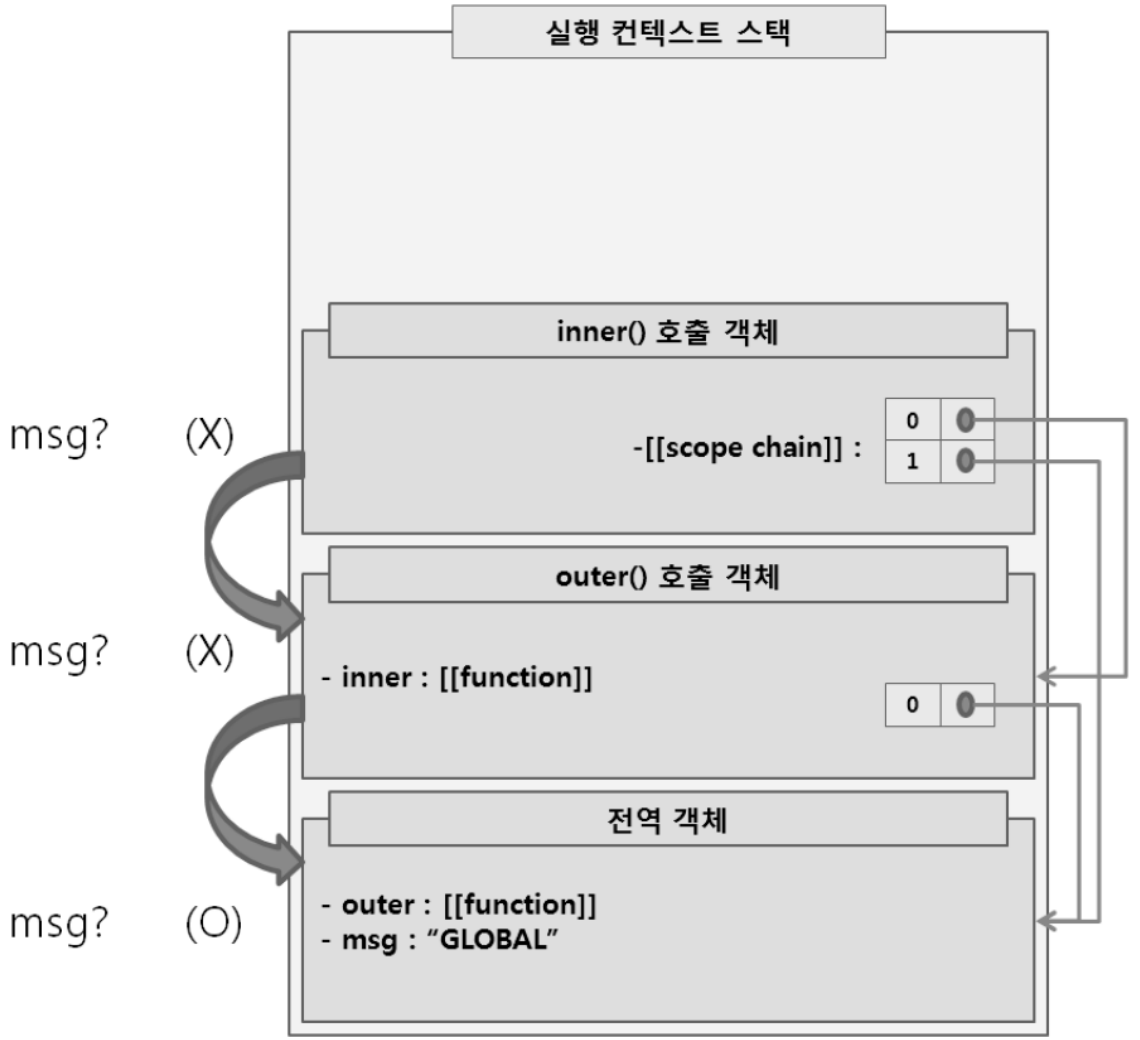
❖ 이전 예제 수정(예제 01-10 : 3,7번 라인 주석처리)

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

5. 스코프와 스코프 체인

❖예제 01-10 실행 구조

- 스코프 체인 생성
  - 자신을 호출한 호출객체에 대한 참조 정보 리스트
  - inner는 outer와 global에 대한 참조 정보를 보유함
- inner() 함수 호출 객체 내부에서 msg를 접근함
  - 자신의 호출객체에는 msg 변수가 없음
  - 스코프체인을 따라가면서 각 호출객체에 msg가 있는지를 찾음
  - global까지 찾았음에도 없다면 변수가 선언되지 않았음을 알리고 오류 발생
- 이 예제에서는 전역의 msg를 접근함



5. 스코프와 스코프 체인

❖호출 객체 단위로 스코프가 결정됨

- 예제 01-11 확인
- 2행의 num과 3행의 num은 동일한 변수를 가리킨다.
- var 키워드를 사용하면 블록 단위 스코프는 존재하지 않음
- let 키워드를 사용하면 블록 단위 스코프를 적용할 수 있음

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

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

5. 스코프와 스코프 체인

❖예제 01-11의 var를 let으로 바꿨을 때 결과

- 예제 01-12
- 2행의 num과 3행의 num은 서로 다른 스코프임
- 따라서 for문 내의 num이 변화하더라도 2행의 num에 영향을 주지 않음

```
function test() {
    let num = 100;
    for (let num = 0; num < 10; num++ ) {
        console.log(num);
    }
    return num;
}

let result = test();
console.log("num : " + result);
```

```
0                                   01-12.html:8
1                                   01-12.html:8
2                                   01-12.html:8
3                                   01-12.html:8
4                                   01-12.html:8
5                                   01-12.html:8
6                                   01-12.html:8
7                                   01-12.html:8
8                                   01-12.html:8
9                                   01-12.html:8
num : 100                           01-12.html:14
>
```

5. 스코프와 스코프 체인

❖호이스팅 과정은 함수 코드 내부에서 스캔한 코드로 진행됨

- 예제 01-13
- 5행의 코드는 절대 실행되지 않지만 호이스팅은 수행됨.
  - 함수 호출 객체에 g라는 변수가 선언되고, undefined 로 초기화됨
  - 따라서 실행해도 오류가 발생하지 않음

```
var g = "GLOBAL";
function test() {
    console.log(g);
    if (false) {
        var g = "TEST";
    }
    console.log(g);
}
test();
```

- 실행 결과 : undefined, undefined

6. 스코프 연습 문제

❖문제 1 : 예제 01-14

```
function test1(a1) {
    a1();
    function a1() {
        console.log("world");
    }
}

test1(function() {
    console.log("hello");
})
```

- 호이스팅 단계에서
  - arguments 및 파라미터 전달에서 1행의 파라미터 a1으로 9행의 익명함수 전달
  - 3행의 선언적 함수 값이 a1에 할당되면서 a1이 변경됨
- 실행 단계
  - 호이스팅 단계가 완료되고나서 함수 내부 코드 실행 --〉 a1() 함수 호출
  - 따라서 결과는 "world"

6. 스코프 연습 문제

❖문제 2 : 예제 01-15

```
var a2 = 1;
function test2() {
    a2 = 10;
    return;
    function a2() {}
}
test2();
console.log(a2);
```

- 호이스팅 단계에서
  - test2() 함수 생성, 전역 변수 a2 선언
  - test2() 호출로 실행 컨텍스트와 호출 객체 생성 --〉 콜 스택에 추가
  - 함수 내부 코드에 대한 호이스팅 단계 진행 --〉 a2 내부 함수 생성
  - a2 변수에 10 할당 --〉 함수가 number 타입의 값으로 변경됨
  - return 문 실행으로 함수 실행 종료
  - 함수 실행이 종료되면서 test2 호출 객체 내부의 a2 함께 삭제
  - 마지막 라인에서 a2 출력 --〉 전역 변수 a2값인 10이 출력

6. 스코프 연습 문제

❖문제 3 : 예제 01-16

- 익명 함수, 즉시 실행 함수 사용

```
var test3 = (function f() {
   function f() { return "hello" }
   return f;
   function f() { return "world" }
})();

console.log(test3());
```

- 즉시 실행함수 호출로 인해 만들어진 호출객체 내부에서 호이스팅 단계가 일어나고 2행, 4행의 선언적 함수가 순차적으로 만들어진다. 호이스팅이 완료되고 나면 23행이 실행되면서 리턴한다.
- 리턴된 값은 1행의 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

❖클로저의 정의

- 외부 함수 안의 내부 함수가 전역에서 참조되고, 내부 함수가 외부 함수안의 지역 변수를 이용할 수 있게 되는 현상 또는 이 때의 내부 함수
- 스코프 체인을 이용해 호출이 완료된 함수의 내부 변수를 참조할 수 있는 방법
- 특정 함수 내의 지역 변수를 외부에서 접근할 수 있도록 하는 내부 함수

❖위의 각 설명은……

- 서로 다른 설명처럼 보이지만 동일한 내용임
- 클로저를 이해하려면 함수 호출 과정, 스코프, 호이스팅 개념을 이해해야 함

❖함수 호출이 완료되면 호출 객체는 가비지 컬렉션 대상이 되어 메모리가 회수됨

- 하지만 호출이 완료된 호출객체가 가비지 컬렉션되지 않는 경우가 있음
- 함수 안에 내부 함수가 포함되어 있고 그 내부함수를 다른 스코프에서 접근할 수 있는 상황

7. 클로저

❖클로저 예시 : 가비지 컬렉션 되지 않는 상황

```
function outer(x) {
    function inner(y) {
        return x+y;
    }
    return inner;
}

var a = outer(4);
var result = a(5);
console.log(result);
```

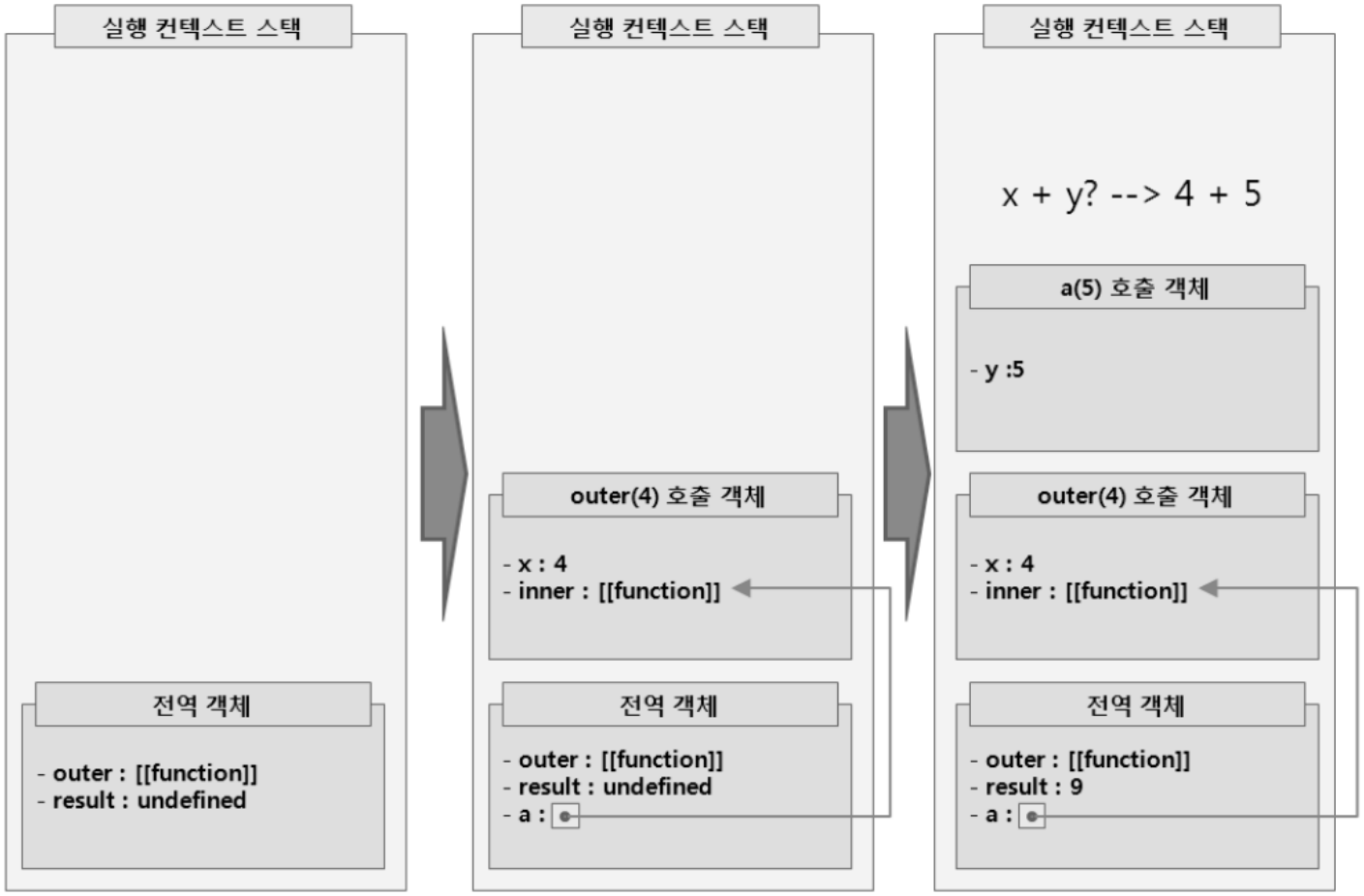
자유 변수

클로저 함수

- outer 함수 호출 객체 내부에 만들어진 자유 변수 x는 클로저 함수를 통해서만 접근할 수 있음

7. 클로저

❖클로저 예시 : 가비지 컬렉션 되지 않는 상황 (이어서)



7. 클로저

❖클로저가 여러개 생성되는 경우 : 예제 01-18

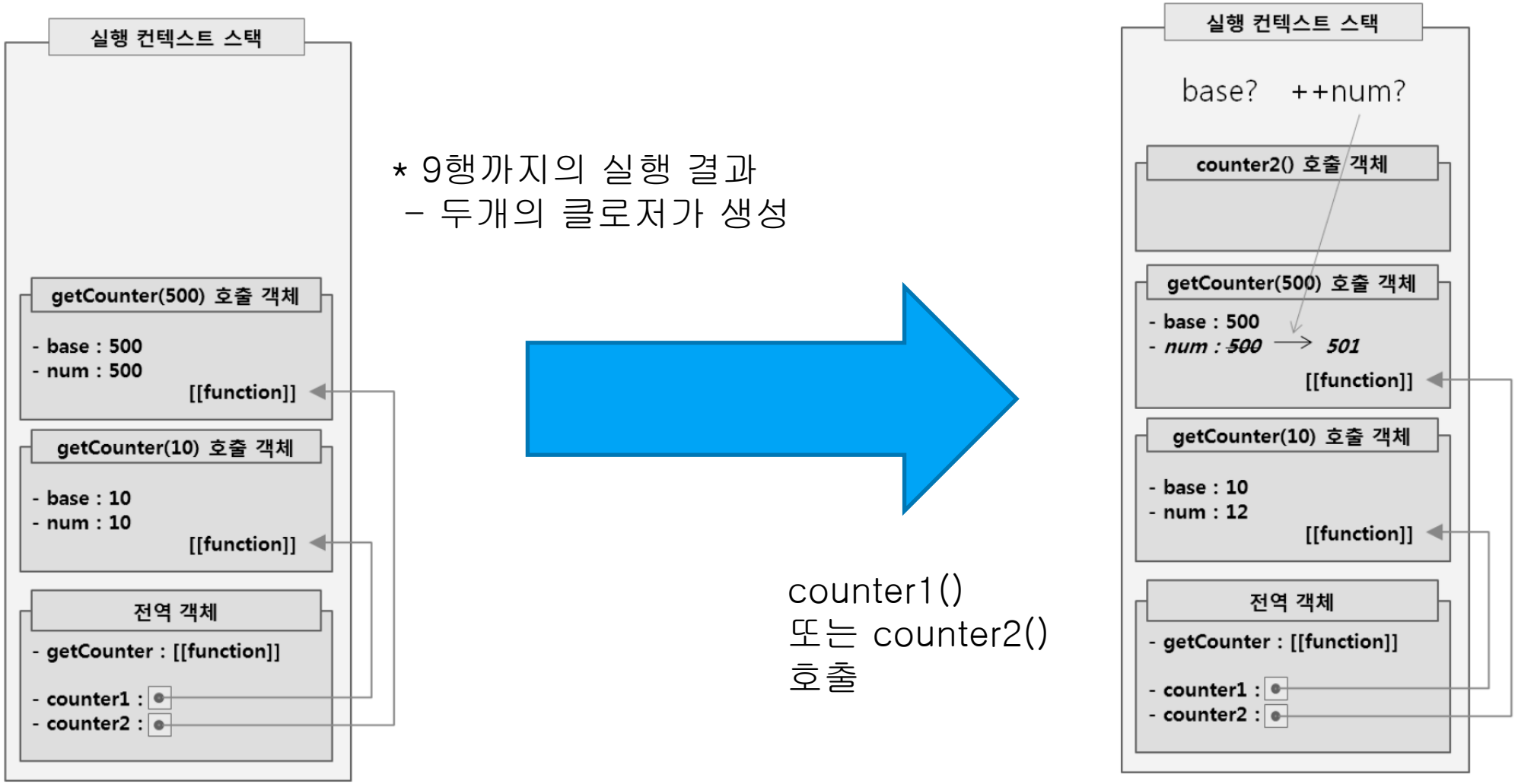
- 단순히 함수의 중첩 구조만 고려해서는 안됨

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

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

console.log(counter1());
console.log(counter1());
console.log(counter2());
console.log(counter2());
console.log(counter2());
```

```
▼ Object i
    base: 10
    count: 11
  ▶ [[Prototype]]: Object

▼ Object i
    base: 10
    count: 12
  ▶ [[Prototype]]: Object

▼ Object i
    base: 500
    count: 501
  ▶ [[Prototype]]: Object

▼ Object i
    base: 500
    count: 502
  ▶ [[Prototype]]: Object

▼ Object i
    base: 500
    count: 503
  ▶ [[Prototype]]: Object
```

7. 클로저

❖클로저가 여러개 생성되는 경우 : 예제 01-18 (이어서)



8. 클로저와 리액트

❖클로저는 리액트에서 어떻게 사용되는가?

- useState 내부가 클로저로 작성되어 있음
- 개발자가 작성하는 Custom Hook들도 클로저를 이용해 만들게 됨
- useState 훅의 Pseudo Code

```
const TestReact = (() => {
    let currentStateKey = 0;
    const states = [];

    const useState = (initialValue) => {
        const key = currentStateKey;

        if (states.length === key) {
          states.push(initialValue);
        }

        const state = states[key];
        const setState = (newState) => {
            states[key] = newState;
            render();
        }
        currentStateKey += 1;
        return [ state, setState ];

    };

    return { useState };
})();
```

8. 클로저와 리액트

❖예제 검토

- src/hooks/useTimer.js

```
import { useEffect, useState } from "react";

const useTimer = (endMessage) => {
    const [timer, setTimer] = useState(10);
    const [handle, setHandle] = useState(null);

    useEffect(()=>{
      if (timer < 1) {
        clearInterval(handle);
        console.log(endMessage);
      }
    }, [timer, endMessage, handle])
```

```
    const startTimer = (timer) => {
      setTimer(timer);
      let tempHandle = setInterval(()=> {
        setTimer((timer) => timer-1);
      }, 1000)
      setHandle(tempHandle);
    }

    const stopTimer = () => {
      clearInterval(handle);
    }

    return { timer, startTimer, stopTimer };
}

export { useTimer };
```

8. 클로저와 리액트

❖예제 검토

- App2.jsx

```
import React, { useState } from 'react';
import { useTimer } from './hooks/useTimer';

const App2 = () => {
    const [seconds1, setSeconds1 ] = useState(10);
    const [seconds2, setSeconds2 ] = useState(15);
    const { timer:timer1, startTimer:startTimer1, stopTimer:stopTimer1 } = useTimer("타이머1 종료");
    const { timer:timer2, startTimer:startTimer2, stopTimer:stopTimer2 } = useTimer("Timer2 is finished");

    const setTimerSeconds1 = (e) => {
        let timeSecs = parseInt(e.target.value, 10);
        timeSecs = isNaN(timeSecs) ? 100 : timeSecs >= 10 ? timeSecs : 10;
        setSeconds1(timeSecs);
    }
    const setTimerSeconds2 = (e) => {
        let timeSecs = parseInt(e.target.value, 10);
        timeSecs = isNaN(timeSecs) ? 100 : timeSecs >= 10 ? timeSecs : 10;
        setSeconds2(timeSecs);
    }
```

8. 클로저와 리액트

```
    return (
        <div>
            <div style={{ border: "solid 1px blue", margin:"10px" }}>
                <h2>타이머1 설정</h2><hr/>
                시간 설정 : <input type="text" value={seconds1} onChange={setTimerSeconds1} />
                <button onClick={()=>startTimer1(seconds1)}>시작</button>
                <button onClick={stopTimer1}>중지</button>
                <hr />
                타이머1 : {timer1}
            </div>
            <div style={{ border: "solid 1px blue", margin:"10px" }}>
                <h2>타이머2 설정</h2><hr/>
                시간 설정 : <input type="text" value={seconds2} onChange={setTimerSeconds2} />
                <button onClick={()=>startTimer2(seconds2)}>시작</button>
                <button onClick={stopTimer2}>중지</button>
                <hr />
                타이머2 : {timer2}
            </div>
        </div>
    );
};

export default App2;
```

8. 클로저와 리액트

❖App2 실행 관련 콜스택 개요도