JavaScript

함수2 – 김근형 강사

함수의 다양한 형태

- 즉시 실행 함수
  - 함수를 정의함과 동시에 바로 실행하는 함수
  - 최소 한 번의 실행만을 필요로 하는 초기화 코드 부분에 사용될 수 있다.

```
(function(인자1, 인자2...){
    함수에서 바로 실행할 로직
})(변수1, 변수2,...);
```

```
(function (name) {
    console.log('This is the immediate function --> ' + name);
})('foo');
```

함수의 다양한 형태

- 내부 함수
  - 함수 내부에서 정의된 함수를 의미
  - 클로저를 생성하거나 부모 함수 코드에서 외부에서의 접근을 막고 독립적인 헬퍼 함수를 구현하는 용도로 사용한다.

```
// parent() 함수 정의
function parent() {
    var a = 100;
    var b = 200;

    // child() 내부 함수 정의
    function child() {
        var b = 300;
        console.log(a); // 100
        console.log(b); // 300
    }
    child();
}
parent();
child(); // Uncaught ReferenceError
```

함수의 다양한 형태

- 내부 함수의 특징
  - 내부 함수는 자신을 둘러싼 부모 함수의 변수에 접근이 가능하다.
  - 내부 함수는 일반적으로 자신이 정의된 부모 함수 내부에서만 호출이 가능하다.
  - 내부 함수를 외부로 리턴하면, 부모 함수 밖에서도 내부 함수를 호출하는 것이 가능하다.

```
// self() 함수
var self = function () {
    console.log('a');
    return function () {
        console.log('b');
    }
}
self = self();  // a
self();  // b
```

```
function parent() {
    var a = 100;

    // child() 내부 함수
    var child = function () {
        console.log(a);
    }

    // child() 함수 반환
    return child;
}

var inner = parent();
inner();  // 100
```

함수의 다양한 형태

- 콜백함수
  - 콜백함수는 함수의 제어권을 다른 대상에게 넘길 때 쓴다.

주기함수 호출 | 인자1: 콜백함수

```
setInterval(function () {
  console.log('1초마다 실행될 겁니다.');
}, 1000);
```

인자2: 주기 (ms)

```
var cb = function() {
  console.log('1초마다 실행될 겁니다.');
};

setInterval(cb, 1000);
```

함수의 다양한 형태

- 콜백함수 만들기
  - 콜백함수는 매개변수에 함수를 호출하여 해당 함수 안에서 실행시키도록 하여 만들 수 있다.

```
function func(callback) {
    console.log( typeof (callback) );
    // 넘어온 함수 실행하기
    callback();
}

function myCall() {
    console.log("myCall 호출됨");
}

func(myCall);
```

함수의 다양한 형태

- 콜백함수 특징

콜백함수의 특징

- 다른 함수(A)의 매개변수로 콜백함수(B)를 전달하면, A가 B의 **제어권을** 갖게 된다.
- 특별한 요청(bind)이 없는 한 A에 **미리 정해진 방식**에 따라 B를 호출한다.
  - 미리 정해진 방식이란 **this**에 무엇을 바인딩할지, **매개변수**에는 어떤 값들을 지정할지, 어떤 **타이밍**에 콜백을 호출할지 등이다.

함수의 다양한 형태

- 콜백 함수와 비동기 처리
  - 비동기 프로그래밍이란, 어떤 작업을 요청한 후 다른 작업을 수행하다가 이벤트가 발생하면 그에 대한 응답을 받아 처리하는 것을 의미한다.

```
function square(x) {
    return x*x;
}

var number = square(2);
console.log(number);  // 4
```

```
function square(x, callback) {
    setTimeout(callback, 100, x*x);
}

var number = 0;
square(2,function(x) {
    number = x;
});
console.log(number); // 0
```

함수의 다양한 형태

- 콜백 함수와 비동기 처리
  - 비동기로 프로그래밍을 할 때, 실행 순서를 신경쓰며 코딩을 해야한다. 위와 같은 이유로, 비동기로 코드들 짤 때, 콜백지옥에 빠지게 된다.

```
function square(x, callback) {
    setTimeout(callback, 100, x*x);
}

square(2,function(x) {
    square(x,function(x2) {
        square(x2,function(x3) {
            console.log(x3);
        });
    });
});  // 256
```

함수의 다양한 형태

- 콜백 함수와 비동기 처리
  - 콜백함수로 비동기 처리를 동기화 하게 되면 함수의 모습이 상당히 이상하게 나오므로 아래와 같이 함수에 이름을 붙여 순차적으로 호출하도록 명시 시켜 처리하는 것이 좋다

```
function square(x, callback) {
    setTimeout(callback, 100, x*x);
}

square(2, firstCallback);

var firstCallback=function (number) {
    square(number, secondCallback);
}

var secondCallback =function (number) {
    square(number, thirdCallback);
}

var thirdCallback =function (number) {
    console.log(number);
}
```

클로저

- 클로저의 정의

A *closure* is the combination of a function and the lexical environment within which that function was declared.

함수 내부에서 생성한 **데이터**와
그 **유효범위**로 인해 발생하는
특수한 **현상 / 상태**

클로저

- 클로저의 특징

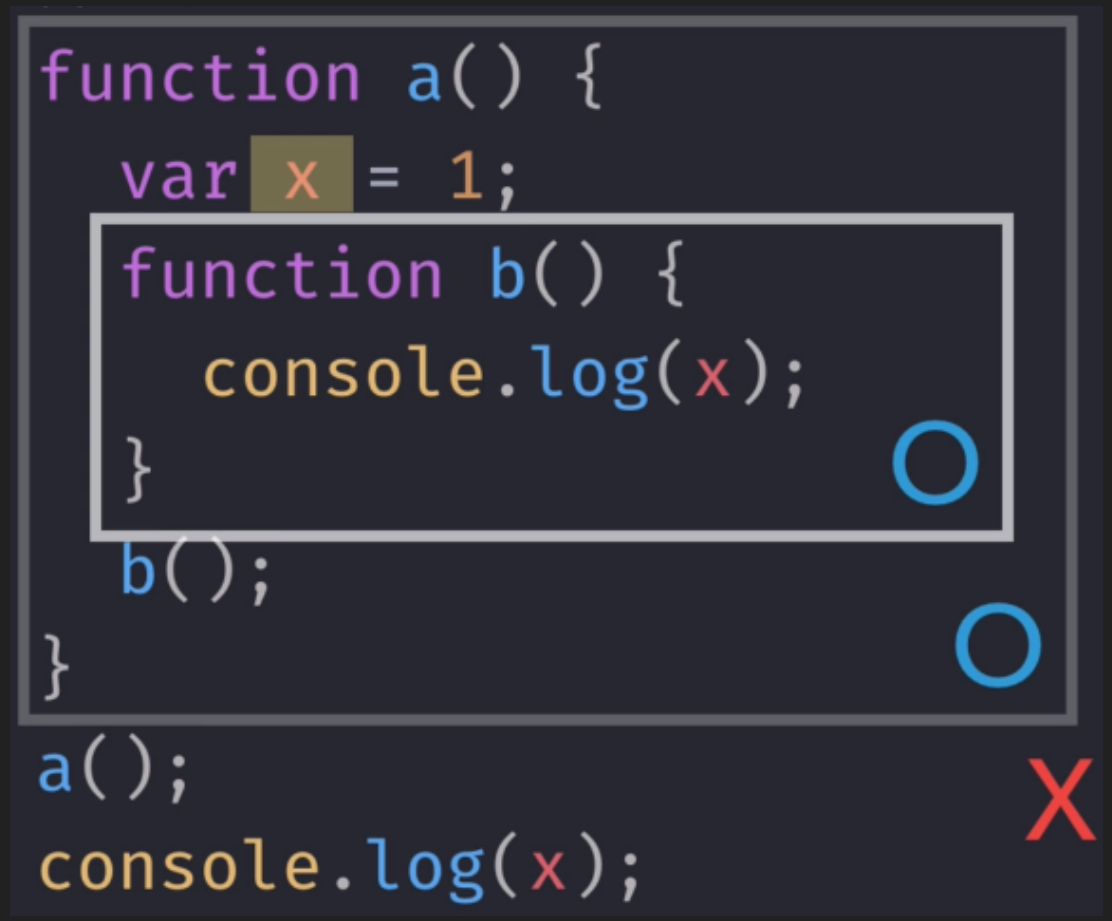
접근 권한 제어

지역변수 보호

데이터 보존 및 활용

클로저

- 클로저 예제
  - var x를 함수 내부에서는 접근이 가능하지만 밖에서는 접근이 불가능하다.



클로저

- 클로저 예제
  - 외부에서 x의 값을 얻을 수는 있으나 함수 내부의 값을 바꿀 수는 없다.

```
function a() {
  var x = 1;
  return function b() {
    console.log(x);
  }
}
var c = a();
c();
```

클로저

- 클로저 예제
  - 함수 내부에서 권한을 줄 경우 함수를 통해 외부와의 소통이 가능해진다.
  - 또한 지역변수 보호도 가능해진다.

```
function a() {
  var _x = 1;
  return {
    get x() { return _x; },
    set x(v) { _x = v; }
  }
}
var c = a();
c.x = 10;
```

클로저

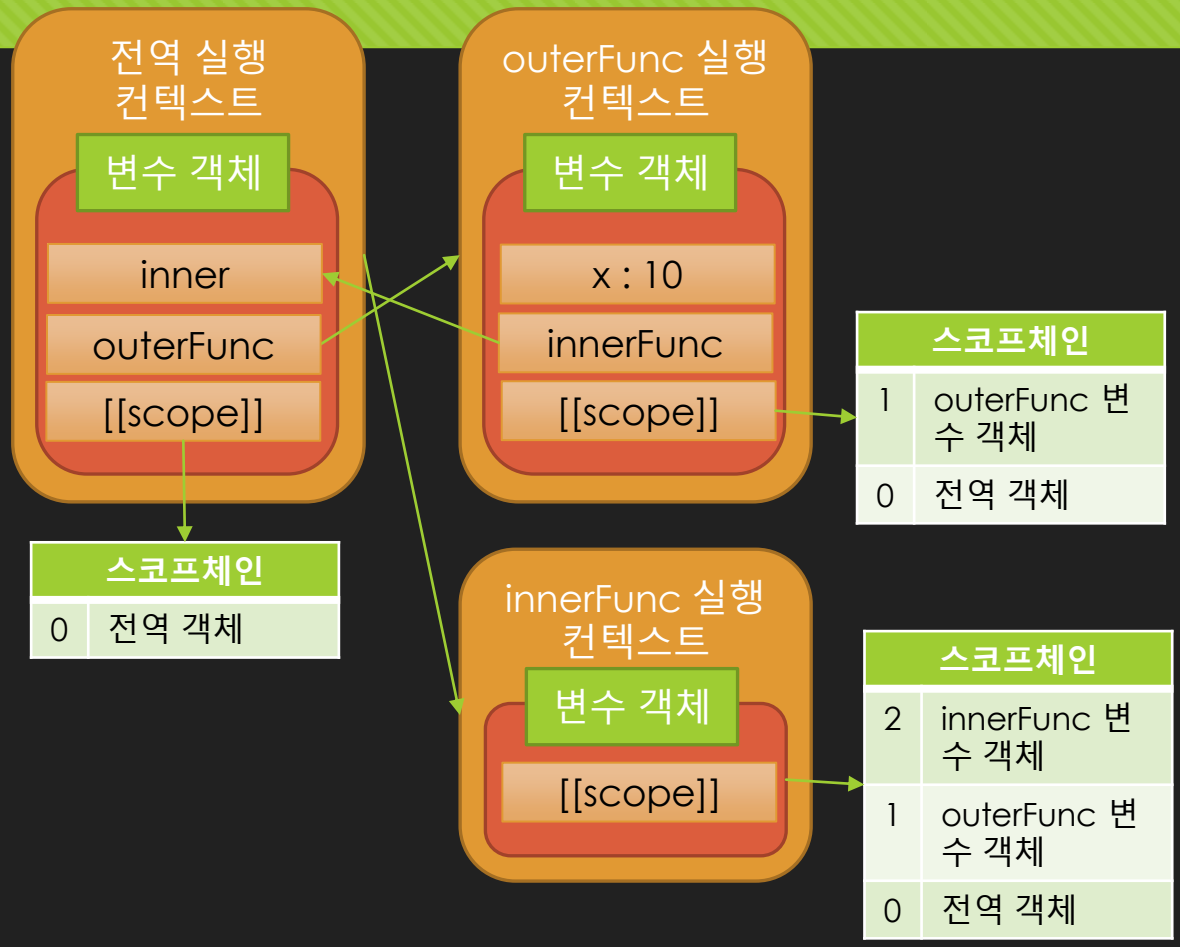
- 스코프 체인으로 바라본 클로저 - 1

```
function outerFunc(){
  var x = 10;
  var innerFunc = function() { console.log(x); }
  return innerFunc;
}

var inner = outerFunc();
inner();
```

자유변수 : 클로저에 의해 참조되는 변수

클로저



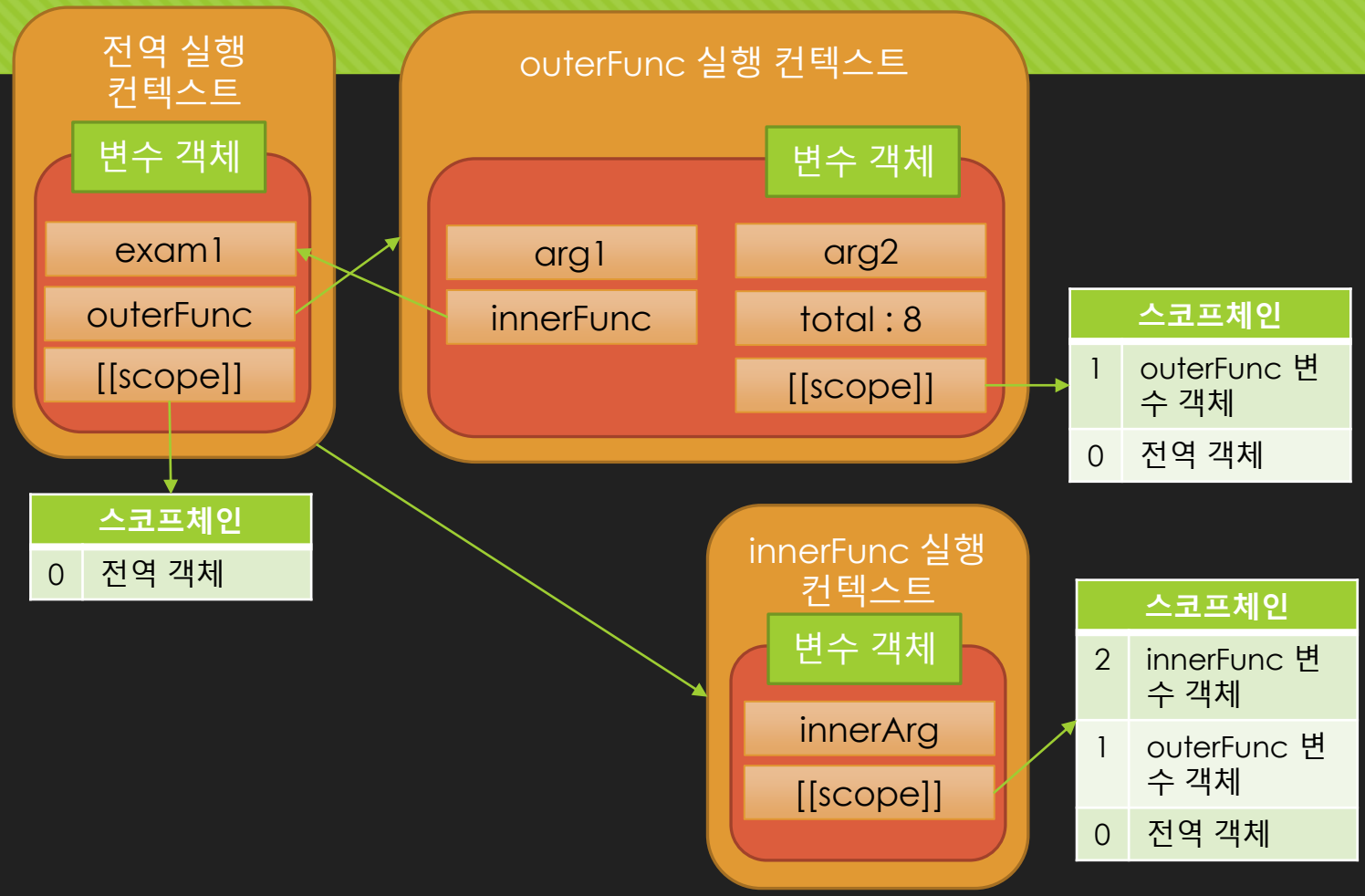
클로저

- 스코프 체인으로 바라본 클로저 - 2

```
function outerFunc(arg1, arg2){
    var local = 8;
    function innerFunc(innerArg){
        console.log((arg1 + arg2)/(innerArg + local));
    }

    return innerFunc;
}

var exam1 = outerFunc( arg1: 2,  arg2: 4);
exam1(2);
```



클로저

- 클로저 동작 흐름

```
function setName(name) {
  return function() {
    return name;
  }
}
var sayMyName = setName('고무곰');
sayMyName();
```

클로저

- 클로저 동작 흐름
  - 스코프는 정의될 때 결정된다.

```
function setCounter() {
  var count = 0;
  return function() {
    return ++count;
  }
}
var count = setCounter();
count();
```

setCounter 정의 [GLOBAL > setCounter]

setCounter 실행

setCounter 스코프에 count 변수 선언 및 0 할당

익명함수 정의 및 반환 [GLOBAL > setCounter > 익명]

반환된 익명함수를 변수 count에 할당

count 실행

익명함수 스코프에서 count 탐색
-> setCounter 스코프에서 count 탐색
-> count에 1을 증가시킨 값을 반환.

클로저

- 클로저를 활용할 때 유의사항
  - 클로저의 프로퍼티 값이 쓰기 가능하므로 그 값이 여러 번 호출로 항상 변할 수 있음에 유의해야 한다.

```
function outerFunc(argNum) {
  var num = argNum;
  return function(x) {
    num += x;
    console.log('num: ' + num );
  }
}

var exam = outerFunc( argNum: 40);

exam(5);
exam(-10);
```

```
num: 45
num: 35
```

클로저

- 클로저를 활용할 때 유의사항
  - 하나의 클로저가 여러 함수 객체의 스코프 체인에 들어가 있는 경우도 있다.

```
function func(){
    var x = 1;
    return {
        func1 : function(){ console.log(++x); },
        func2 : function(){ console.log(-x); }
    };
};

var exam = func();
exam.func1();
exam.func2();
```

```
2
-2
```

클로저

- 클로저를 활용할 때 유의사항
  - 루프 안에서 클로저를 활용할 때는 주의할 것

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    setTimeout( handler: function () {
      console.log(i);
    }, timeout: i * 1000);
  }
};
countSeconds( howMany: 3);
```

```
4
4
4
```

```
function countSeconds(howMany) {
  for (var i = 1; i <= howMany; i++) {
    (function () {
      var currentI = i;
      setTimeout( handler: function () {
        console.log(currentI);
      }, timeout: currentI * 1000);
    }());
  }
};
countSeconds( howMany: 3);
```

```
1
2
3
```