HOISTING

* 스코프 내부에 선언된 모든 변수들은 최상위에서 선언된것으로 간주
* TDZ(Temporal Dead Zone)
* 코프의 시작 지점부터 초기화 시작 지점까지의 구간
* let & const
* var 는 Funtion-scpoed (if, for, while 등 내에서 선언 외부 접근가능) / 일반 함수는 X
* let & const 는 block-scoped (블록내에서만 유지 외부에서 접근 불가)

변수 생성과정 선언>초기화>할당

var = 선언과 초기화가 동시에 이루어짐 > 할당

let = 선언과 초기화가 분리

const = 선언과 할당이 동시 // 상수

생성자

* 생성자 함수는 첫글자를 대문자로 작명

심볼

sumbol() //

전역 심볼

symbol.for() //

클로저

**클로저 (Closures)**

클로저는 다른말로 익명 함수라고 한다.

보통의 함수는 외부에서 인자를 받아 로직을 처리한다. 하지만 클로저는 자신을 둘러싼 context 내의 변수에 접근할 수 있다. 즉, 일급 객체 함수(first-class functions)의 개념을 이용해 외부 범위의 변수를 함수 내부로 바인딩하는 기술이다.

특이한점은 자신을 둘러싼 외부 함수가 종료되더라도 이 값이 유지된다는 것 (기억한다는 것)이다. 함수에서 사용하는 값들은 클로저가 생성되는 시점에서 정의되고 함수 자체가 복사되어 따로 존재하기 떄문이다.

예를 들면 이러하다.

function startAt(x){

function incrementBy(y){

return x + y

}

return incrementBy

}

var closure1 = startAt(1)

var closure2 = startAt(2)

각각의 closure1과 closure2는 y+1과 y+2 값을 반환하는 함수의 역할이다.

이 코드에서 incrementBy 함수는 코드 블록에 속해 있으므로, 인수인 x에 대한 접근이 가능하다. startAt 함수는 x의 값이나 참조를 복사한 클로저를 반환한다.

# 1. 클로저(closure)의 개념

클로저(closure)는 자바스크립트에서 중요한 개념 중 하나로 자바스크립트에 관심을 가지고 있다면 한번쯤은 들어보았을 내용이다. [execution context](https://poiemaweb.com/js-execution-context)에 대한 사전 지식이 있으면 이해하기 어렵지 않은 개념이다. 클로저는 자바스크립트 고유의 개념이 아니라 함수를 일급 객체로 취급하는 함수형 프로그래밍 언어(Functional Programming language: 얼랭(Erlnag), 스칼라(Scala), 하스켈(Haskell), 리스프(Lisp)…)에서 사용되는 중요한 특성이다.

클로저는 자바스크립트 고유의 개념이 아니므로 ECMAScript 명세에 클로저의 정의가 등장하지 않는다. 클로저에 대해 [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures)은 아래와 같이 정의하고 있다.

“A closure is the combination of a function and the lexical environment within which that function was declared.”  
클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경(Lexical environment)과의 조합이다.

무슨 의미인지 잘 와닿지 않는다. 위 정의에서 중요한 키워드는 “함수가 선언됐을 때의 렉시컬 환경(Lexical environment)”이다.

말이 무척이나 난해하니 우선 예제부터 살펴보자. “The Art of Computer Programming”의 저자 [도널드 커누스](https://ko.wikipedia.org/wiki/%EB%8F%84%EB%84%90%EB%93%9C_%EC%BB%A4%EB%88%84%EC%8A%A4)의 말처럼 우리 모두는 자신의 힘으로 발견한 내용을 가장 쉽게 익힌다.

function outerFunc() {

var x = 10;

var innerFunc = function () { console.log(x); };

innerFunc();

}

outerFunc(); // 10

함수 outerFunc 내에서 내부함수 innerFunc가 선언되고 호출되었다. 이때 내부함수 innerFunc는 자신을 포함하고 있는 외부함수 outerFunc의 변수 x에 접근할 수 있다. 이는 함수 innerFunc가 함수 outerFunc의 내부에 선언되었기 때문이다.

스코프는 함수를 호출할 때가 아니라 함수를 어디에 선언하였는지에 따라 결정된다. 이를 [**렉시컬 스코핑(Lexical scoping)**](https://poiemaweb.com/js-scope#7-%EB%A0%89%EC%8B%9C%EC%BB%AC-%EC%8A%A4%EC%BD%94%ED%94%84)라 한다. 위 예제의 함수 innerFunc는 함수 outerFunc의 내부에서 선언되었기 때문에 함수 innerFunc의 상위 스코프는 함수 outerFunc이다. 함수 innerFunc가 전역에 선언되었다면 함수 innerFunc의 상위 스코프는 전역 스코프가 된다.

함수 innerFunc가 함수 outerFunc의 내부에 선언된 내부함수이므로 함수 innerFunc는 자신이 속한 렉시컬 스코프(전역, 함수 outerFunc, 자신의 스코프)를 참조할 수 있다. 이것을 [실행 컨텍스트](https://poiemaweb.com/js-execution-context)의 관점에서 설명해보자.

내부함수 innerFunc가 호출되면 자신의 실행 컨텍스트가 실행 컨텍스트 스택에 쌓이고 변수 객체(Variable Object)와 스코프 체인(Scope chain) 그리고 this에 바인딩할 객체가 결정된다. 이때 스코프 체인은 전역 스코프를 가리키는 전역 객체와 함수 outerFunc의 스코프를 가리키는 함수 outerFunc의 활성 객체(Activation object) 그리고 함수 자신의 스코프를 가리키는 활성 객체를 순차적으로 바인딩한다. 스코프 체인이 바인딩한 객체가 바로 렉시컬 스코프의 실체이다.

내부함수 innerFunc가 자신을 포함하고 있는 외부함수 outerFunc의 변수 x에 접근할 수 있는 것, 다시 말해 상위 스코프에 접근할 수 있는 것은 렉시컬 스코프의 레퍼런스를 차례대로 저장하고 있는 실행 컨텍스트의 **스코프 체인**을 자바스크립트 엔진이 검색하였기에 가능한 것이다. 좀더 자세히 설명하면 아래와 같다.

1. innerFunc 함수 스코프(함수 자신의 스코프를 가리키는 활성 객체) 내에서 변수 x를 검색한다. 검색이 실패하였다.
2. innerFunc 함수를 포함하는 외부 함수 outerFunc의 스코프(함수 outerFunc의 스코프를 가리키는 함수 outerFunc의 활성 객체)에서 변수 x를 검색한다. 검색이 성공하였다.

이번에는 내부함수 innerFunc를 함수 outerFunc 내에서 호출하는 것이 아니라 반환하도록 변경해 보자.

function outerFunc() {

var x = 10;

var innerFunc = function () { console.log(x); };

return innerFunc;

}

/\*\*

\* 함수 outerFunc를 호출하면 내부 함수 innerFunc가 반환된다.

\* 그리고 함수 outerFunc의 실행 컨텍스트는 소멸한다.

\*/

var inner = outerFunc();

inner(); // 10

함수 outerFunc는 내부함수 innerFunc를 반환하고 생을 마감했다. 즉, 함수 outerFunc는 실행된 이후 콜스택(실행 컨텍스트 스택)에서 제거되었으므로 함수 outerFunc의 변수 x 또한 더이상 유효하지 않게 되어 변수 x에 접근할 수 있는 방법은 달리 없어 보인다. 그러나 위 코드의 실행 결과는 변수 x의 값인 10이다. 이미 life-cycle이 종료되어 실행 컨텍스트 스택에서 제거된 함수 outerFunc의 지역변수 x가 다시 부활이라도 한 듯이 동작하고 있다. 뭔가 특별한 일이 일어나고 있는 것 같다.

**이처럼 자신을 포함하고 있는 외부함수보다 내부함수가 더 오래 유지되는 경우, 외부 함수 밖에서 내부함수가 호출되더라도 외부함수의 지역 변수에 접근할 수 있는데 이러한 함수를 클로저(Closure)라고 부른다.**

다시 MDN의 정의로 돌아가 보자.

“A closure is the combination of a function and the lexical environment within which that function was declared.”  
클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경(Lexical environment)과의 조합이다.

위 정의에서 말하는 “함수”란 반환된 내부함수를 의미하고 “그 함수가 선언될 때의 렉시컬 환경(Lexical environment)”란 내부 함수가 선언됐을 때의 스코프를 의미한다. 즉, **클로저는 반환된 내부함수가 자신이 선언됐을 때의 환경(Lexical environment)인 스코프를 기억하여 자신이 선언됐을 때의 환경(스코프) 밖에서 호출되어도 그 환경(스코프)에 접근할 수 있는 함수**를 말한다. 이를 조금 더 간단히 말하면 **클로저는 자신이 생성될 때의 환경(Lexical environment)을 기억하는 함수다**라고 말할 수 있겠다.

클로저에 의해 참조되는 외부함수의 변수 즉 outerFunc 함수의 변수 x를 **자유변수(Free variable)**라고 부른다. 클로저라는 이름은 자유변수에 함수가 닫혀있다(closed)라는 의미로 의역하면 자유변수에 엮여있는 함수라는 뜻이다.

[실행 컨텍스트](https://poiemaweb.com/js-execution-context)의 관점에 설명하면, 내부함수가 유효한 상태에서 외부함수가 종료하여 외부함수의 실행 컨텍스트가 반환되어도, 외부함수 실행 컨텍스트 내의 **활성 객체(Activation object)**(변수, 함수 선언 등의 정보를 가지고 있다)는 **내부함수에 의해 참조되는 한 유효**하여 내부함수가 **스코프 체인**을 통해 참조할 수 있는 것을 의미한다.

즉 외부함수가 이미 반환되었어도 외부함수 내의 변수는 이를 필요로 하는 내부함수가 하나 이상 존재하는 경우 계속 유지된다. 이때 내부함수가 외부함수에 있는 변수의 복사본이 아니라 실제 변수에 접근한다는 것에 주의하여야 한다.

클래스

**Promise**

* 비동기 처리를 위한 객체

  //Promise

  //new Promise 생성자가 반환하는 객체

  //state:pendung(대기)

  //result:undefined

  //resolve(value) ->

  //state: fullfilled(이행)

  //result:value

  //reject(error)

  //state: rejected(거부)

  //result:error

  const pr = new Promise((resolve, reject) => {

    setTimeout(() => {

      resolve('OK') //반환값

    }, 3000)

    //success = resolve

    // failed = reject

  });

  //then 이후 사용 하는 함수 catch(reject인경우에만 실행). finally

  pr.then(function (result) { }, //이행되었을 시 실행 result = 'OK'

    function (err) { }); // 거부 되었을 시 실행

  //같은 코드

  pr.then((result) => { }).catch((err) => { }).finally(() => { }) //finally 이행//거부 항상 실행

  //Promise 없이 일반 함수 로 작성 시

  function Fn1(callback) {

    console.log('Fn1')

    callback();

  };

  function Fn2(callback) {

    console.log('Fn2')

    callback();

  };

  function Fn3(callback) {

    console.log('Fn3')

    callback();

  };

  Fn1(Fn2(Fn3()))//... callback hell

  //Promise 처리

  const f1 = () => {

    return new Promise((res, rej) => {

      //code

    })

  }

  const f2 = (param) => {

    return new Promise((res, rej) => {

      //code

    })

  }

  const f3 = (param) => {

    return new Promise((res, rej) => {

      //code

    })

  }

  f1()

    .then(res => f2(res))

    .then(res => f3(res))

    .catch((err) => {

      //에러

    }).finally(() => {

      //완료

    })

    //에러가 실행되면 뒤에 함수는 실행 X

    //Promise.all(배열)

    Promise.all([f1(), f2(), f3()]).then((res)=>{

      //세작업이 동시 실행 및 모두 완료된 후 then 실행

      //중간에 오류가 뜨면 반환값이 없음

      //중간에 누락되는 페이지가 없어야 하는 경우에 사용

    })

    //Promise.race(배열) - 경주

    //하나라도 완료되면 뒤에 작업은 X

**async & await**

    //async를 사용하면 Promise를 반환한다

    async function getPromise() {

        return new Promise((res) => { }, (rej) => { })

    }

    //async 사용함수는 뒤에 then 사용가능

    getPromise().then((value) => { })

    //await 키워드는 async 내에서만 사용가능

    async function setAwait() {

        await getPromise(); //await 키워드 오른쪽엔 Promise 형식 // 그 Promise가 처리 될때 까지 기다림

    }

    async function order() {

        const result1 = await fn1();

        const result2 = await fn2(result1);

        const result3 = await fn3(result2);

        console.log(`ENd ${result3}`)

        //or

        const result\_all = await Promise.all([fn1(), fn2(), fn3()]);

        console.log(`ENd ${result3}`)

    }

    async function order() {

        //오류 발생 처리 try catch

        try {

            const result1 = await fn1();

            const result2 = await fn2(result1);

            const result3 = await fn3(result2);

            console.log(`ENd ${result3}`)

        } catch (err) {

            console.log(`ERR`)

        }

        console.log(`ENd ${result3}`)

    }

**Generator**

* 함수의 실행을 중간에 멈췄다가 재개

    //함수옆에 \* 사용으로 제너레이터 선언

    function\* fn() {

        // yield 키워드로 함수의 사용을 멈춤

        console.log('1')

        yield 1; //data

        console.log('2')

        yield 2;

        yield 3;

        return 'end'

    }

    //next 메소드 사용

    //가장 가까운 yield 키워드 만날때까지 실행후 멈춤

    fn().next();

    //-> yield 1을 만날때까지 함수 실행 후 데이터 객체 반환

    //반환된 객체는 value:, done: property를 가짐

    //value: yield 의 r-value 값

    //done: 함수코드가 끝났는지 안끝났는지 bool값 반환

    //done : false = 실행할 함수가 남음

    //done : true = 모든 함수 실행 완료

    //메소드

    //return(), throw()

    fn().return() // 그 즉시 완료

    fn().throw()   //건너뛰기

    //iterable

    //symbol.iterator 메서드가 있다

    //symbol.iterator 는 iterator를 반환해야 한다

    //iterator

    //next 메서드를 가진다

    //next메소드는 value와 done속성을 가진 객체를 반환

    //작업이 끝나면 done 은 true가 된다