EliceAcademy

JavaScript : 3주차

실행 컨텍스트와 this

목차

프로토타입에 대해 이해할 수 있다.

실행 컨텍스트의 원리를 알 수 있다.

this의 의미에 대해 설명 할 수 있다.

01

프로토타입

프로토타입이란?

자바스크립트의 프로토 타입을 기반으로 상속을 구현한다
부모 객체의 프로퍼티 또는 메소드를 상속받아 사용할 수 있게 한다.
이 부모 객체를 Prototype(프로토타입) 객체 혹은 그냥 Prototype(프로토타입) 이라고 한다
최상위 Prototype 객체는 Object

모든 객체는 자신의 프로토타입 객체를 가리키는
[[Prototype]] 인터널 슬롯(internal slot) 을 갖으며 상속을 위해 사용된다.

함수도 객체이므로 [[Prototype]] 인터널 슬롯을 갖는다.
그런데 함수 객체는 일반 객체와는 달리 prototype 프로퍼티가 추가로 있다

참고:
https://poiemaweb.com/js-prototype

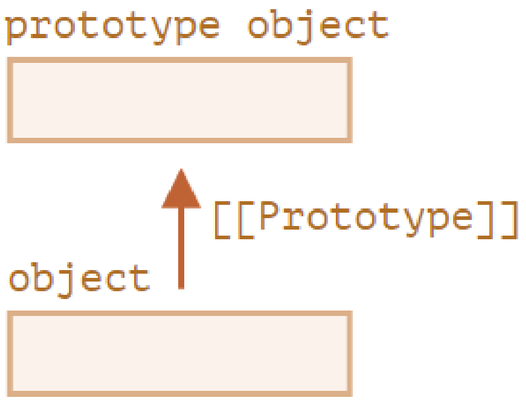
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

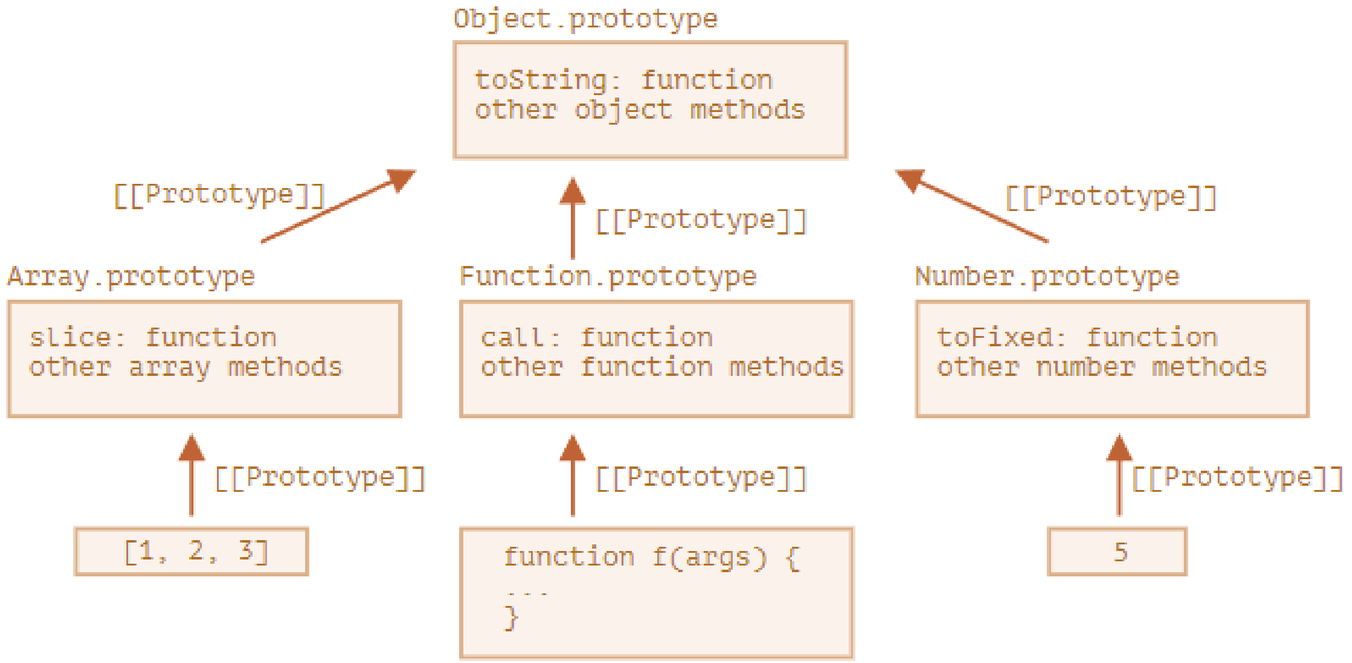
EliceAcademy

프로토타입과 프로토타입 체인

객체간 상속의 연결고리는
프로토타입 체인으로 연결 되어 있다

클래스도 프로토타입 체인구조 이다
단지 문법적으로 깔끔하게 표현한 것

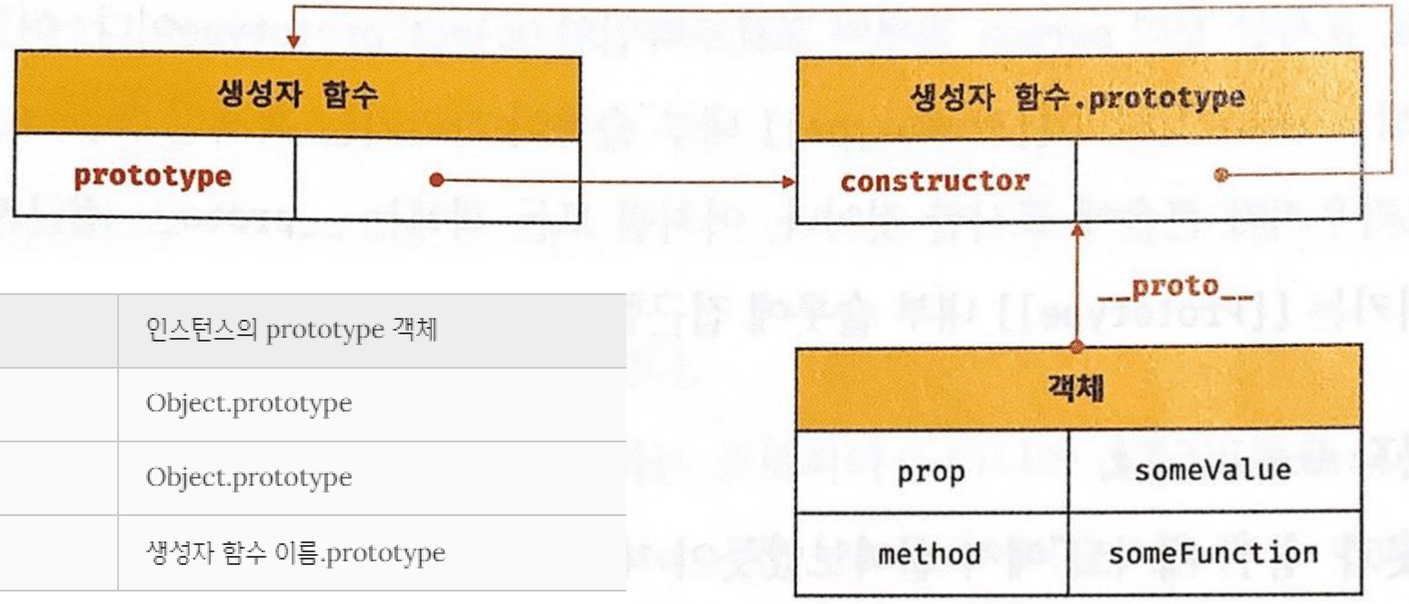




EliceAcademy

prototype 과 _ _proto_ _

```
const obj1 = {};
const obj2 = new Object();
const obj3 = new Person();
```



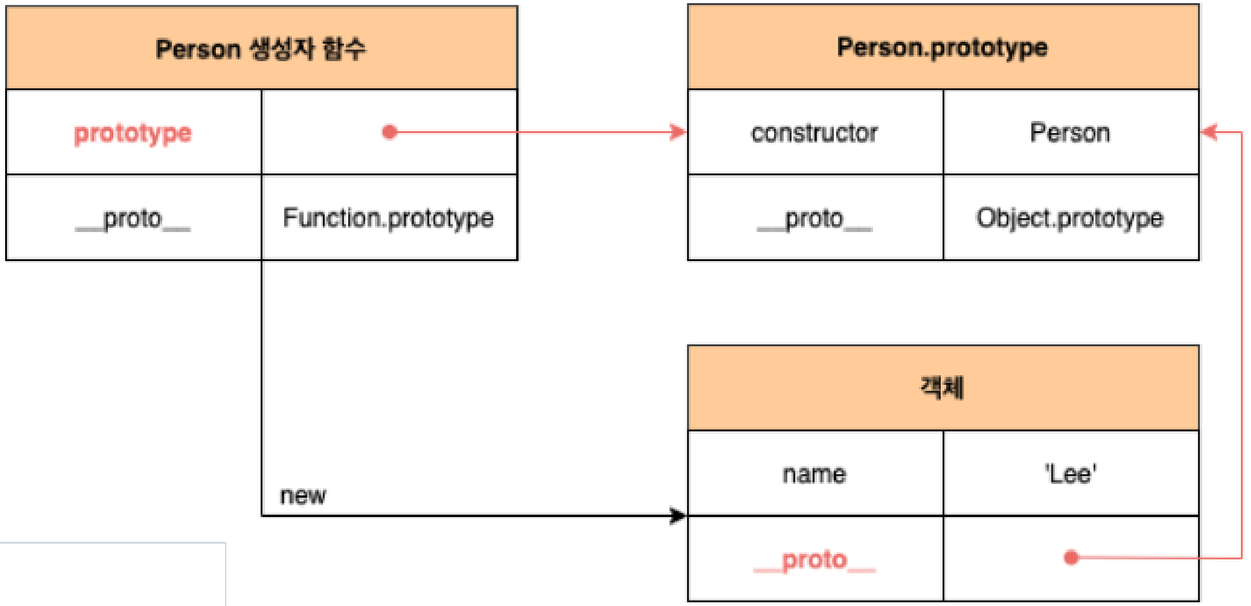
객체 생성 방식	엔진의 객체 생성	인스턴스의 prototype 객체
객체 리터럴	Object() 생성자 함수	Object.prototype
Object() 생성자 함수	Object() 생성자 함수	Object.prototype
생성자 함수	생성자 함수	생성자 함수 이름.prototype

그림 19-3 객체와 프로토타입과 생성자 함수는 서로 연결되어 있다.

property 는 함수 객체만 가지고 있고 이는 함수 객체가 생성자로 사용될 때 이 함수를 통해 생성될 객체의 부모 역할을 하는 객체(프로토타입 객체)를 가리킨다.

EliceAcademy

__proto__와 prototype 차이



구분	소유	값	사용 주체	사용 목적
proto 접근자 프로퍼티	모든 객체	프로토타입의 참조값	모든 객체	객체가 자신의 프로토타입에 접근 또는 교체하기 위해 사용
prototype 프로퍼티	constructor	프로토타입의 참조값	생성자 함수	생성자 함수가 자신이 생성할 객체(인스턴스)의 프로토타입을 할당하기 위해 사용

프로퍼티 어트리뷰트 : 데이터 프로퍼티

프로퍼티 어트리뷰트	프로퍼티 디스크립터 객체의 프로퍼티	설명
[[Value]]	Value	■ 프로퍼티 키를 통해 프로퍼티 값에 접근하면 반환되는 값이다. ■ 프로퍼티 키를 통해 프로퍼티 값을 변경하면 [[Value]]에 값을 재할당한다. 이때 프로퍼티가 없으면 프로퍼티를 동적으로 생성하고 생성된 프로퍼티의 [[Value]]에 값을 저장한다.
[[Writable]]	Writable	■ 프로퍼티 값의 변경 가능 여부를 나타내며 불리언 값을 갖는다. ■ [[Writable]]의 값이 false인 경우 해당 프로퍼티의 [[Value]]의 값을 변경할 수 없는 읽기 전용 프로퍼티가 된다.
[[Enumerable]]	Enumerable	■ 프로퍼티의 열거 가능 여부를 나타내며 불리언 값을 갖는다. ■ [[Enumerable]]의 값이 false인 경우 해당 프로퍼티는 for…in 문이나 Object.keys 메서드 등으로 열거할 수 없다.
[[Configurable]]	Configurable	■ 프로퍼티의 재정의 가능 여부를 나타내며 불리언 값을 갖는다. ■ [[Configurable]]의 값이 false인 경우 해당 프로퍼티의 삭제, 프로퍼티 어트리뷰트 값의 변경이 금지된다. 단, [[Writable]]이 true인 경우 [[Value]]의 변경과 [[Writable]]을 false로 변경하는 것은 허용된다

프로퍼티 어트리뷰트 : 접근자 프로퍼티

프로퍼티 어트리뷰트	프로퍼티 디스크립터 객체의 프로퍼티	설명
[[Get]]	get	■ 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 읽을 때 호출되는 접근자 함수이다. 즉, 접근자 프로퍼티 키로 프로퍼티 값에 접근하면 프로퍼티 어트리뷰트 [[Get]]의 값, 즉 getter 함수가 호출되고 그 결과가 프로퍼티 값으로 반환된다.
[[Set]]	set	■ 접근자 프로퍼티를 통해 데이터 프로퍼티의 값을 저장할 때 호출되는 접근자 함수이다. 즉, 접근자 프로퍼티 키로 프로퍼티 값을 저장하면 프로퍼티 어트리뷰트 [[Set]]의 값, 즉 setter 함수가 호출되고 그 결과가 프로퍼티 값으로 저장된다.
[[Enumerable]]	enumerable	■ 데이터 프로퍼티의 [[Enumerable]] 과 같다.
[[Configurable]]	configurable	■ 데이터 프로퍼티의 [[Configurable]] 과 같다.

02

실행컨텍스트

EliceAcademy

호이스팅(hosting)

참고:
https://developer.mozilla.org/ko/docs/Glossary/Hoisting

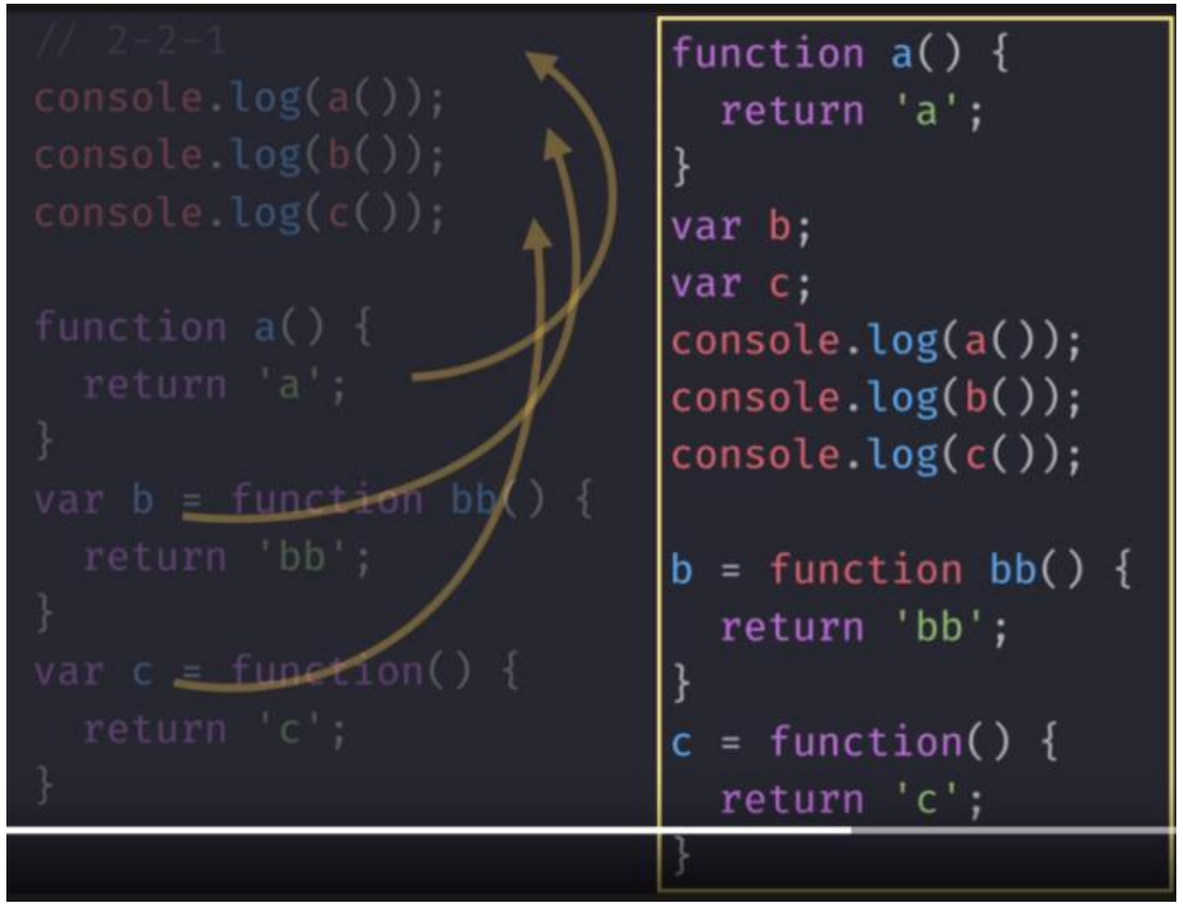
변수 선언(var) / 함수 선언문을 위로 끌어 올려서 해당
함수 유효 범위의 최상단에 선언한다

호이스팅은 스코프 단위로 발생

var : 선언과 초기값(undefined) 동시에
호이스팅되서 미리 설정됨
할당은 런타임에 적용

let : 선언, 초기값 할당은 런타임
const : 선언과 초기값 할당이 동시에 됨

let과 const는 호이스팅 되지만 TDZ에서 관리



TDZ

변수 할당 전에 사용하면 에러가 발생한다
=> 코드 예측 가능하고 잠재적인 에러를 줄일 수 있다.

선언은 되어있지만, 초기화가 되지 않아 이를 위한 자리가 메모리에 준비되어 있지 않은 상태


```
let age = 30;

function showAge(){
    console.log(age);

    let age = 20;
}

showAge();
```

실행 컨텍스트 excution context

실행 컨텍스스트는 소스코드를 실행하는데 필요한 환경을 제공하고 코드의 실행 결과를 실제로 관리하는 영역 => 모든 코드는 실행 컨텍스트를 통해 실행되고 관리 된다.

렉시컬 환경(Lexical Environment)
스코프를 구분하여 식별자를 등록하고 관리하는 저장소 역할
환경 레코드(Environment Record)
스코프에 포함된 식별자를 등록하고 등록된 식별자에 바인딩된 값을 관리하는 저장소
외부 렉시컬 환경에 대한 참조 (Quter Lexical Environment Reference)
외부 렉시컬 환경에 대한 참조는 상위 스코프 (함수를 호출한 곳)

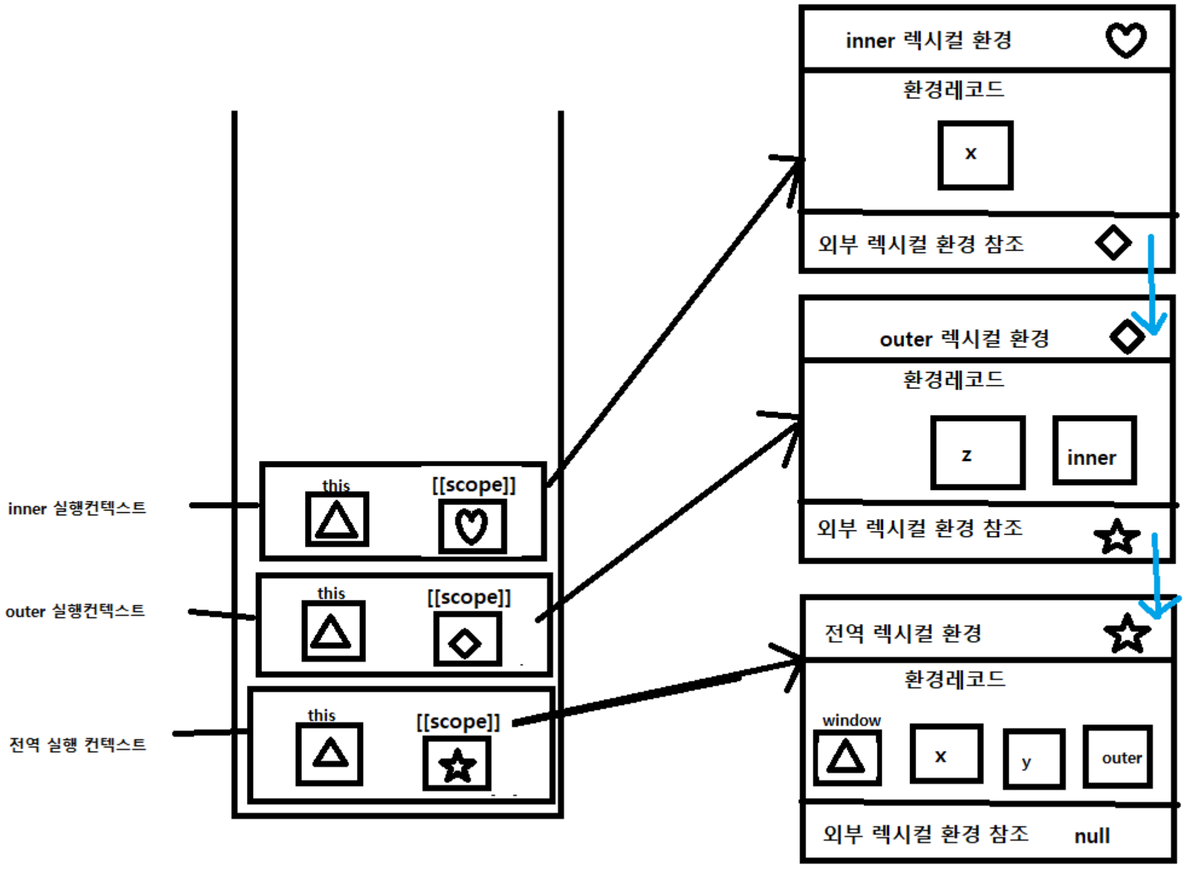
EliceAcademy

렉시컬 환경과 스코프 체인

```
let x = 'global x';
let y = 'global y';

function outer() {
    let z = 'local z';
    console.log('outer 영역이야 ');
    function inner() {
        let x = 'local x';
        console.log('inner 영역이야');
    }
    inner();
}

outer();
```



클로저

클로저는 함수와 그 함수가 선언됐을 때의 렉시컬 환경(Lexical environment)과의 조합이다.

내부 함수에서 외부 함수 스코프에 접근할 수 있는 것
closures : 폐쇠 -> 내부 정보를 은닉하고 공개 함수를 통한 데이터 조작
캡슐화와 정보 은닉 => 클래스의 private 필드 이전에 사용 방식

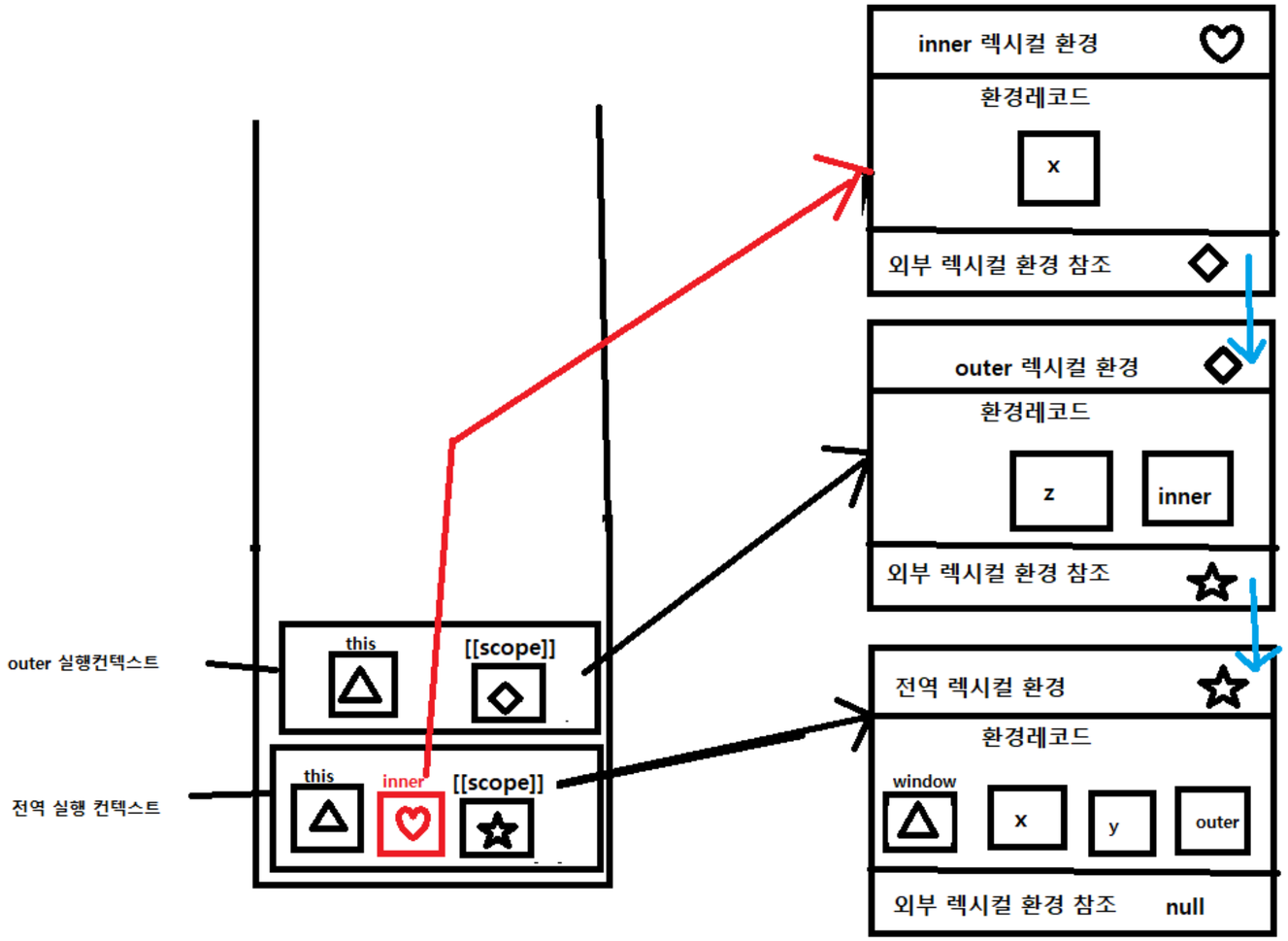
EliceAcademy

클로저

```
let x = 'global x';
let y = 'global y';

function outer() {
    let z = 'local z';
    console.log('outer 영역이야 ');
    function inner() {
        let x = 'local x';
        console.log('inner 영역이야');
    }
    //inner();
    return inner;
}

const inner = outer();
inner();
```



EliceAcademy

03

this

this 바인딩

자바스크립트에서 this는 동적으로 결정됨
다른 객체지향 언어에서는 this는 항상 정적으로 자신의 인스턴스 주소

바인딩 => 식별자와 값을 연결하는 과정
this 바인딩 => this와 this가 가리킬 객체를 바인딩하는 것

일반 함수(중첩 함수, 콜백 함수) this -> window (global) 전역 객체
생성자(메서드) 함수 this -> 생성된 인스턴스 객체
이벤트 리스너 this -> 이벤트 객체
화살표 함수 : 렉시컬(lexical) this : 상위 스코프의 this 객체

렉시컬 스코프

동적 스코프 (dynamic scope)
함수가 호출되는 시점에 동적으로 상위 스코프를 결정

정적 스코프(static scope) 또는 렉시컬 스코프(lexical scope)
함수를 어디서 정의 함에 따라 상위 스코프를 결정

대부분 프로그래밍 언어가 렉시컬 스코프

apply,call, bind 이용한 this 바인딩

apply (this로 사용할 객체, arguments 리스트(배열 or 유사배열객체))
call (this로 사용할 객체 , arguments 인수 리스트(,로 구분하여 전달))
bind (this로 사용할 객체)

apply, call

본질적은 기능은 함수를 호출하는 것 : **일회성**

함수를 호출하면서 첫 번째 인수로 전달한 특정 객체를 호출한 함수의 this에 바인딩
두번째 인수를 함수에 전달하는 방식만 다를 뿐 동일하게 작동

bind : 함수의 this값 **영구적** 변경 가능

본질적인 기능은 함수를 호출하진 않고 this로 사용할 객체만 전달(리턴)

메서드의 this와 메서드 내부의 중첩 함수 or 콜백 함수의 this가 불일치에 사용

화살표 함수 특징

1. 함수를 축약해서 가독성이 올라감
2. 생성자 함수로 사용 불가능 => prototype 객체가 없다
3. 함수 내부 arguments 객체도 없다
4. this가 자동으로 상위 스코프의 this로 정적 바인딩 됨

strict mode

참고
https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Strict_mode

리엑트와 같은 프레임워크 사용시 기본적으로 엄격 모드
자바스크립트에서는 엄격모드가 따로 설정이 안되서 수동으로
코드나 함수 맨 위에 ‘use strict’ 이라고 적어야 함 => 함수 단위는 비추천
var 키워드 생략 불가능
this 는 보통 객체에 쓰이기 때문에 함수 내부 this 자동 전역객체 x => undefined