4/29
4/30
5/1
5/4 : 1h

13. 스코프

TABLE OF CONTENTS

1. 스코프란?
2. 스코프의 종류
    2.1. 전역과 전역 스코프
    2.2. 지역과 지역 스코프
3. 스코프 체인
    3.1. 스코프 체인에 의한 변수 검색
    3.2. 스코프 체인에 의한 함수 검색
4. 함수 레벨 스코프
5. 렉시컬 스코프

#1. 스코프란?

스코프(Scope, 유효범위)는 자바스크립트를 포함한 모든 프로그래밍 언어의 기본적이며 중요한 개념이다. 스코프의 이해가 부족하면 다른 개념을 이해하기 어려울 수 있다. 더욱이 자바스크립트의 스코프는 다른 언어의 스코프와 구별되는 특징을 가지므로 주의가 필요하다. 그리고 var 키워드로 선언한 변수와 let 또는 const 키워드로 선언한 변수의 스코프도 다르게 동작한다. 스코프는 변수 그리고 함수와 깊은 관련이 있다. 스코프는 식별자, 함수와 관련 있음

우리는 스코프를 이미 경험했다. 함수의 매개변수는 함수 몸체 내부에서만 참조할 수 있고 함수 몸체 외부에서는 참조할 수 없다고 했다. 이것은 매개변수를 참조할 수 있는 유효한 범위, 즉 매개변수의 스코프가

함수 몸체 내부로 한정되기 때문이다.

JAVASCRIPT

```
function add(x, y) {
  // 매개변수는 함수 몸체 내부에서만 참조할 수 있다.
  // 즉, 매개변수의 스코프(유효범위)는 함수 몸체 내부이다.
  console.log(x, y); // 2 5
  return x + y;
}

add(2, 5);

// 매개변수는 함수 몸체 내부에서만 참조할 수 있다.
console.log(x, y); // ReferenceError: x is not defined
```

변수는 코드의 가장 바깥 영역뿐만 아니라 코드 블록이나 함수 몸체 내에서도 선언할 수 있다. 이때 코드 블록이나 함수는 중첩될 수 있다.

JAVASCRIPT

```
var var1 = 1; // 코드의 가장 바깥 영역에서 선언된 변수

if (true) {
  var var2 = 2; // 코드 블록 내에서 선언된 변수
  if (true) {
    var var3 = 3; // 중첩된 코드 블록 내에서 선언된 변수
  }
}

function foo() {
  var var4 = 4; // 함수 내에서 선언된 변수

  function bar() {
    var var5 = 5; // 중첩된 함수 내에서 선언된 변수
  }
}

console.log(var1); // 1
console.log(var2); // 2
console.log(var3); // 3
```

```
console.log(var4); // ReferenceError: var4 is not defined
console.log(var5); // ReferenceError: var5 is not defined
```

M.T : 함수 내에서 선언된 변수는 함수 몸체 내에서만 유효하므로

변수는 자신이 선언된 위치에 의해 자신이 유효한 범위, 즉 다른 코드가 변수 자신을 참조할 수 있는 범위가 결정된다. 변수 뿐만 아니라 모든 식별자가 그렇다. 다시 말해, **모든 식별자(변수 이름, 함수 이름, 클래스 이름 등)는 자신이 선언된 위치에 의해 다른 코드가 식별자 자신을 참조할 수 있는 유효 범위가 결정된다. 이를 스코프(Scope, 유효범위)라 한다. 즉, 스코프는 식별자가 유효한 범위를 말한다.**

아래 예제가 어떻게 동작할지 생각해보자.

JAVASCRIPT

식별자 x가 객체의 프로퍼티가된다 .

```
var x = 'global';
function foo() {
  var x = 'local';

  console.log(x); // ①

}


foo();

console.log(x); // ②
```

이 시점에서는 var x =로컬 선언 안함. 하면 손해임.여기서는 나머지 두 개의 선언문을 선언함.
이 시점에서 전역 스코프라는 실체를 만듦 그리고 foo라는 식별자에 함수 값까지 할당
이 때 함수 몸체 안에 있는 선언문을 실행. 그리고 foo의 지역 스코프가 만들어짐.
함수는 정의한다고 반드시 호출하는 것은 아님.

첫째 console(객체)는 foo 지역 안에서 찾음.
둘째 log(프로퍼티 키)는 프로토타입 체인에서 찾음.
console앞에 window. 이 생략되어 있다고 봐도 됨.
셋째 x를 찾음.
1 정답 : local
콘솔 다음에는 사실 리턴이 있음. 뭐를 리턴? undefined를 리턴한다.

식별자는 스코프 체인에서 찾고, 프로퍼티는 프로토타입에서 찾는다
-> 이 두 개가 협력하여 코드를 실행한다

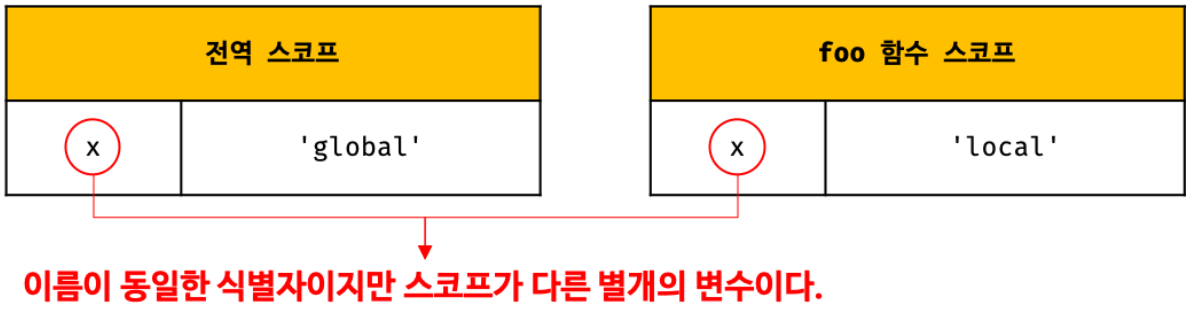
코드의 가장 바깥 영역과 함수 foo 내부에 같은 이름을 갖는 변수 x를 선언하였고 ①과 ②에서 변수 x를 참조한다. 이때 자바스크립트 엔진은 이름이 같은 두개의 변수 중에서 어떤 변수를 참조해야 할 것인지를 결정해야 한다. 자바스크립트 엔진은 스코프를 통해 어떤 변수를 참조해야 할 것인지를 결정한다. 즉, 스코프란 자바스크립트 엔진이 **식별자를 검색할 때 사용하는 규칙**이라고도 할 수 있다.

자바스크립트 엔진은 코드를 실행할 때, 코드의 문맥(Context)를 고려한다. 코드가 어디서 실행되며 주변에 어떤 코드들이 있는지에 따라 위 예제의 ①과 ②처럼 동일한 코드도 다른 결과를 만들어 낸다.

코드의 문맥(Context)과 환경(Environment)

"코드가 어디서 실행되며 주변에 어떤 코드들이 있는지"를 환경(Environment)이라고 부른다. 즉, 코드의 문맥(Context)은 환경들로 이루어진다. 이를 구현한 것이 "실행 컨텍스트(Execution context)"이며 모든 코드는 실행 컨텍스트에서 평가되고 실행된다. 스코프는 실행 컨텍스트와 깊은 관련이 있다. 이에 대해서는 "23. 실행 컨텍스트"에서 자세히 살펴보도록 하자.

위 예제에서 코드의 가장 바깥 영역에 선언된 변수 x는 어디서든 참조할 수 있다. 하지만 함수 foo 내부에서 선언된 변수 x는 함수 foo 내부에서만 참조할 수 있고 함수 foo 외부에서는 참조할 수 없다. 이때 두 개의 변수 x는 식별자 이름이 동일하지만 자신이 유효한 범위, 즉 스코프가 다른 별개의 변수이다.



스코프는 네임스페이스이다.

만약 스코프라는 개념이 없다면 같은 이름을 갖는 변수는 충돌을 일으키므로 프로그램 전체에서 하나밖에 사용할 수 없다.

"4.1 변수란 무엇인가? 왜 필요한가?"에서 살펴본 식별자(identifier)에 대해 다시 한번 생각해보자. 변수나 함수의 이름과 같은 식별자는 어떤 값을 구별하여 식별해낼 수 있는 고유한 이름을 말한다. 사람을 고유한 이름으로 구별하듯이 값도 사람이 이해할 수 있는 언어로 지정한 고유한 식별자인 변수 이름에 의해 구별하여 참조할 수 있다.

식별자는 어떤 값을 구별할 수 있어야 하므로 유일(unique)해야 한다. 따라서 식별자인 변수 이름은 중복될 수 없다. 즉, 하나의 값은 유일한 식별자에 연결(Name binding)되어야 한다.

예를 들어 컴퓨터의 파일명은 하나의 파일을 구별하여 식별할 수 있는 식별자다. 식별자인 파일명은 유일해야 한다. 하지만 우리는 컴퓨터를 사용할 때 하나의 파일명 만을 사용하지 않는다. 식별자인 파일명을 중복해서 사용할 수 이유는 이유는 폴더(디렉터리)라는 개념이 있기 때문이다. 만약 폴더가 없다면 파일명은 유일해야 한다. 컴퓨터 전체를 통틀어 하나의 파일명 만을 사용해야 한다면 파일명을 만드는 것이 무척이나 번거로울 것이다.

이와 마찬가지로 프로그래밍 언어에서는 스코프(유효 범위)를 통해 식별자인 변수 이름의 충돌을 방지하여 같은 이름의 변수를 사용할 수 있도록 한다. 스코프 내에서 식별자는 유일해야 하지만 다른 스코프에는 같은 이름의 식별자를 사용할 수 있다.

> var **키워드로 선언한 변수의 중복 선언**
> var 키워드로 선언된 변수는 같은 스코프 내에서 중복 선언이 허용된다. 이는 의도치 않게 변수값이 재할당되어 변경되는 부작용을 발생시킨다.

JAVASCRIPT

```
function foo() {
  var x = 1;
  // var 키워드로 선언된 변수는 같은 스코프 내에서 중복 선언을 허용한다.
  // 아래 변수 선언문은 자바스크립트 엔진에 의해 var 키워드가 없는 것처럼 동작한다.
  var x = 2;
  console.log(x); // 2
}
foo();
```

-> var x = 2; 로 하든 x=2;로 하든 같다는 말

하지만 let이나 const 키워드로 선언된 변수는 같은 스코프 내에서 중복 선언을 허용하지 않는다.

JAVASCRIPT

```
function bar() {
  let x = 1;
  // let이나 const 키워드로 선언된 변수는 같은 스코프 내에서 중복 선언을 허용하지 않는다.
  let x = 2; // SyntaxError: Identifier 'x' has already been declared
}
bar();
```

#2. 스코프의 종류

코드는 전역(global)과 지역(local)으로 구분할 수 있다.

구분	설명	스코프	변수
전역	코드의 가장 바깥 영역	전역 스코프	전역 변수
지역	함수 몸체 내부	지역 스코프	지역 변수

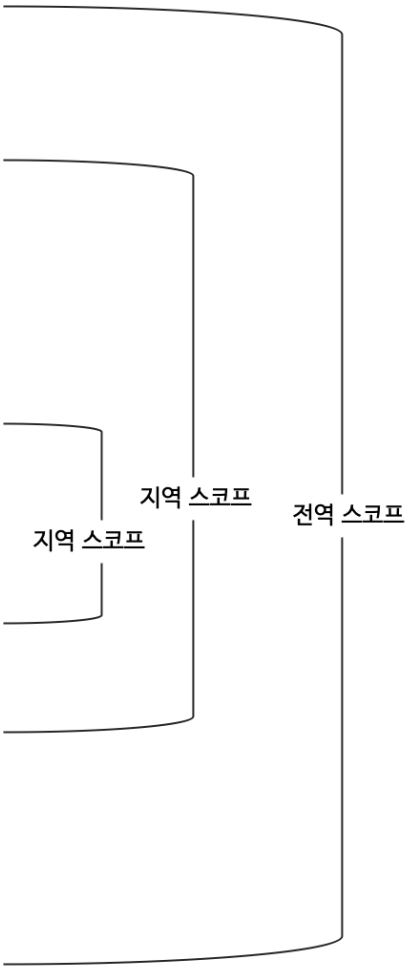
※ 주의 :그냥 {}의 블록문이 아니라 함수만 된다는 것!!

이때 변수는 자신이 선언된 위치(전역 또는 지역)에 의해 자신이 유효한 범위인 스코프가 결정된다. 즉, 전역에서 선언된 변수는 전역 스코프를 갖는 전역 변수이고, 지역에서 선언된 변수는 지역 스코프를 갖는 지역 변수이다.

2.1. 전역과 전역 스코프

아래 예제를 살펴보자.

```
var x = "global x";
var y = "global y";

function outer() {
  var z = "outer's local z";

  console.log(x); // ① global x
  console.log(y); // ② global y
  console.log(z); // ③ outer's local y

  function inner() {
    var x = "inner's local x";

    console.log(x); // ④ inner's local x
    console.log(y); // ⑤ global y
    console.log(z); // ⑥ outer's local z
  }

  inner();
}

outer();

console.log(x); // ⑦ global
console.log(z); // ⑧ ReferenceError: foo is not defined
```



전역 스코프와 지역 스코프

전역이란 코드의 가장 바깥 영역을 말한다. 전역은 전역 스코프(global scope)를 만든다. 전역에 변수를 선언하면 전역 스코프를 갖는 전역 변수(global variable)가 된다. **전역 변수는 어디서든지 참조할 수 있다.**

위 예제에서 코드 가장 바깥 영역인 전역에서 선언된 변수 x와 변수 y는 전역 변수이다. 전역 변수는 어디서든지 참조할 수 있으므로 함수 내부에서도 참조할 수 있다.

2.2. 지역과 지역 스코프

지역이란 **함수 몸체 내부**를 말한다. 지역은 지역 스코프(local scope)를 만든다. 지역에 변수를 선언하면 지역 스코프를 갖는 지역 변수(local variable)가 된다. 지역 변수는 자신이 선언된 지역과 하위 지역(중첩

함수)에서만 참조할 수 있다. 다시 말해 **지역 변수는 자신의 지역 스코프와 하위 지역 스코프에서 유효하다.**

위 예제(그림 12-1)에서 함수 outer 내부에서 선언된 변수 z는 지역 변수이다. 지역 변수 z는 자신의 지역 스코프인 함수 outer 내부와 하위 지역 스코프인 함수 inner 내부에서 참조할 수 있다. 지역 변수 z를 전역에서 참조하면 참조 에러가 발생한다.

함수 inner 내부에서 선언된 변수 x도 지역 변수이다. 지역 변수 x는 자신의 지역 스코프인 함수 inner 내부에서만 참조할 수 있다. 지역 변수 x를 전역 또는 함수 inner 내부 이외의 지역에서 참조하면 참조 에러가 발생한다.

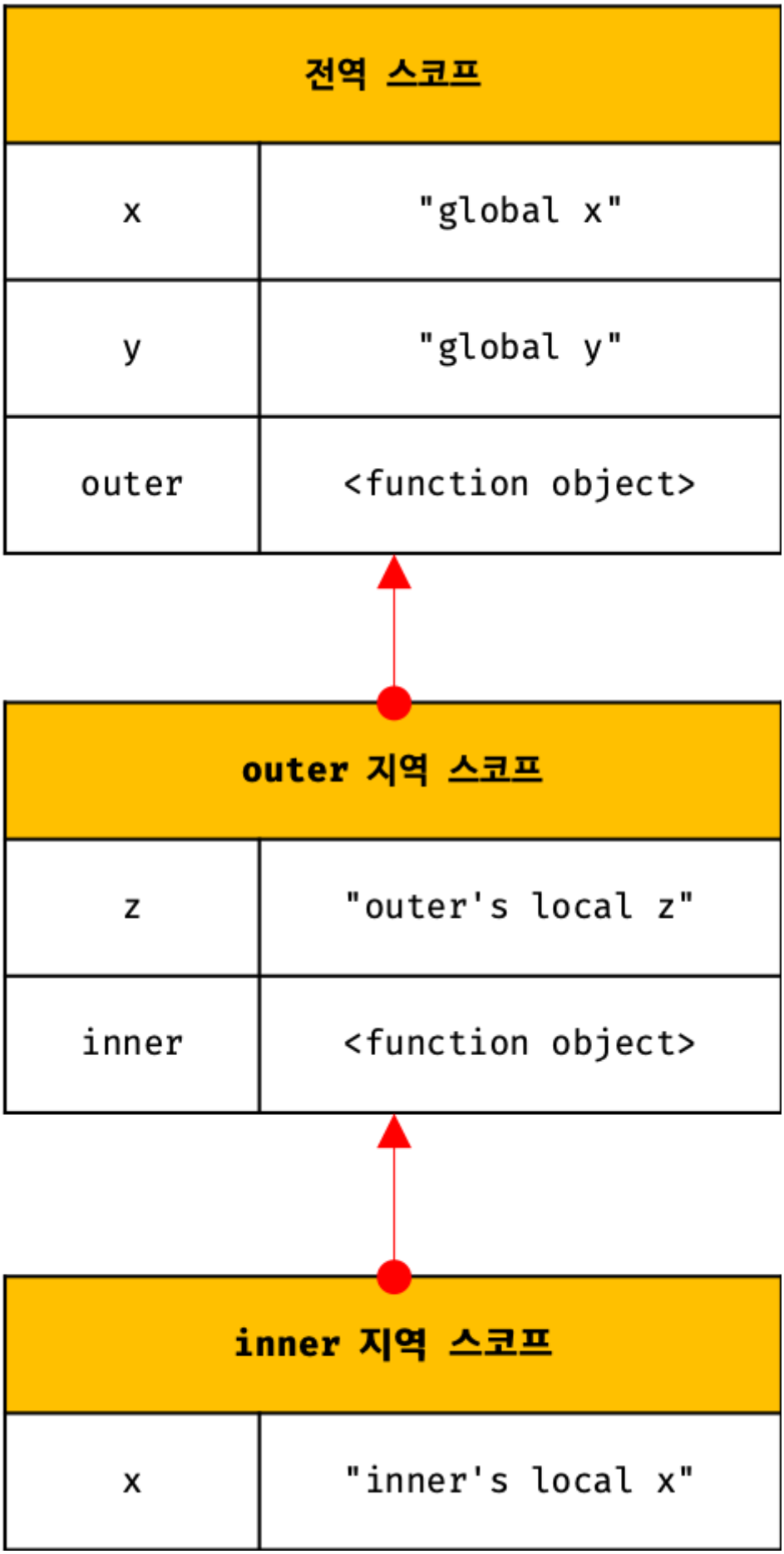
그런데 함수 inner 내부에서 선언된 변수 x 이외에 이름이 같은 전역 변수 x가 존재한다. 이때 함수 inner 내부에서 변수 x를 참조하면 전역 변수 x를 참조하는 것이 아니라 함수 inner 내부에서 선언된 변수 x를 참조한다. 이는 자바스크립트 엔진이 스코프 체인에 통해 참조할 변수를 검색했기 때문이다.

3. 스코프 체인

함수는 전역에서 정의할 수도 있고 함수 몸체 내부에서 정의할 수도 있다. 함수 몸체 내부에서 함수가 정의된 것을 '함수의 중첩'이라 한다. 그리고 함수 몸체 내부에서 정의한 함수를 '중첩 함수(nested function)', 중첩 함수를 포함하는 함수를 '외부 함수(outer function)'라고 부른다.

함수는 중첩될 수 있으므로 함수의 지역 스코프도 중첩될 수 있다. 이는 **스코프는 함수의 중첩에 의해 계층적 구조를 갖는다**는 것을 의미한다. 다시 말해 중첩 함수의 지역 스코프는 중첩 함수를 포함하는 외부 함수의 지역 스코프와 계층적 구조를 갖는다. 이때 외부 함수의 지역 스코프를 중첩 함수의 상위 스코프라 한다.

위 예제에서 지역은 outer 함수의 지역과 inner 함수의 지역이 있다. inner 함수는 outer 함수의 중첩 함수이다. 이때 outer 함수가 만든 outer 함수의 지역 스코프는 inner 함수가 만든 inner 함수의 지역 스코프의 상위 스코프이다. 그리고 outer 함수의 지역 스코프의 상위 스코프는 전역 스코프이다. 이러한 계층적 구조를 그림으로 나타내보면 아래와 같다.



스코프 체인 **-> '단방향 linked list'**

이처럼 모든 스코프는 하나의 계층적 구조로 연결되며 모든 지역 스코프의 최상위 스코프는 전역 스코프이다. 이렇게 스코프가 계층적으로 연결된 것을 **스코프 체인**(Scope chain)이라 부른다. 위 그림에서 스코

프 체인은 최상위 스코프인 전역 스코프, 전역에서 선언된 함수 outer의 지역 스코프, 함수 outer 내부에서 선언된 함수 inner의 지역 스코프로 이루어진다.

변수를 참조할 때, 자바스크립트 엔진은 스코프 체인을 통해 변수를 참조하는 코드의 스코프에서 시작하여 상위 스코프 방향으로 이동하며 선언된 변수를 검색한다. 이를 통해 상위 스코프에서 선언한 변수를 하위 스코프에서도 참조할 수 있다.

스코프 체인은 물리적인 실체로 존재한다. 자바스크립트 엔진은 코드를 실행하기 앞서 위 그림과 유사한 자료 구조(렉시컬 환경 Lexical Environment)를 실제로 생성한다.

> **렉시컬 환경**(Lexical Environment)
> 스코프 체인은 실행 컨텍스트(Execution Context)의 렉시컬 환경(Lexical Environment)을 단방향으로 연결(Channing)한 것이다. 전역 렉시컬 환경은 코드가 로드되면 곧바로 생성되고 함수의 렉시컬 환경은 함수가 호출되면 곧바로 생성된다. 이에 대해서는 “22. 실행 컨텍스트”에서 자세히 살펴보자.

변수 선언이 실행되면 변수 식별자가 이 자료 구조(렉시컬 환경)에 등록되고, 변수의 할당이 일어나면 이 자료 구조의 변수 식별자에 해당하는 값을 변경한다. 변수의 검색도 이 자료 구조 상에서 이루어진다.

3.1. 스코프 체인에 의한 변수 검색

위 예제(그림 12-1)의 ④, ⑤, ⑥을 살펴보자. 이를 통해 자바스크립트 엔진이 스코프 체인을 통해 어떻게 변수를 찾아내는지 이해할 수 있다.

④ 변수 x를 참조하는 코드의 스코프인 inner 함수의 지역 스코프에서 변수 x가 선언되었는지 검색한다. 함수 inner 내에는 선언된 변수 x가 존재한다. 따라서 검색된 변수를 참조하고 검색을 종료한다.

⑤ 변수 y를 참조하는 코드의 스코프인 inner 함수의 지역 스코프에서 변수 y가 선언되었는지 검색한다. 함수 inner 내에는 변수 y의 선언이 존재하지 않으므로 상위 스코프인 함수 outer의 지역 스코프로 이동한다. 함수 outer 내에도 변수 y의 선언이 존재하지 않으므로 또 다시 상위 스코프인 전역 스코프로 이동한다. 전역 스코프에는 변수 y의 선언이 존재한다. 따라서 검색된 변수를 참조하고 검색을 종료한다.

⑥ 변수 z를 참조하는 코드의 스코프인 inner 함수의 지역 스코프에서 변수 z가 선언되었는지 검색한다. 함수 inner 내에는 변수 z의 선언이 존재하지 않으므로 상위 스코프인 함수 outer의 지역 스코프로 이동한다. 함수 outer 내에는 변수 z의 선언이 존재한다. 따라서 검색된 변수를 참조하고 검색을 종료한다.

이처럼 자바스크립트 엔진은 스코프 체인을 따라 변수를 참조하는 코드의 스코프에서 상위 스코프 방향으로 이동하며 선언된 변수를 검색한다. 절대 하위 스코프로 내려가면 식별자를 검색하는 일은 없다. 이는

상위 스코프에서 유효한 변수는 하위 스코프에서 자유롭게 참조할 수 있지만 하위 스코프에서 유효한 변수를 상위 스코프에서 참조할 수 없다는 것을 의미한다.

스코프 체인으로 연결된 스코프의 계층적 구조는 부자 관계로 이루어진 상속(Inheritance)과 유사하다. 상속을 통해 부모의 자산을 자식이 자유롭게 사용할 수 있지만 자식의 자산을 부모가 사용할 수는 없다. 스코프 체인도 마찬가지 개념이다.

3.2. 스코프 체인에 의한 함수 검색

아래 예제를 살펴보자. 전역에서 정의된 foo 함수와 bar 함수 내부에서 정의된 foo 함수가 있다.

JAVASCRIPT

```
// 전역 함수
function foo() {
  console.log('global function foo');
}

function bar() {
  // 중첩 함수
  function foo() {
    console.log('local function foo');
  }

  foo(); // ①
}

bar();
```

```
> function bar(){
      function foo(){
          console.log('local function foo');
      }
      foo();
  }
< undefined
```

'local function foo'가 출력되어야 할 거 같은데 undefined가 반환되는 이유는 function bar는표현식이 아님->표현식이 아닌 문은 완료값인 undefined로 출력되기 때문

"12.4.1 함수 선언문"과 "12.4.3 함수 생성 시점과 함수 호이스팅"에서 살펴보았듯이 함수 선언문으로 함수를 정의하면 자바스크립트 엔진에 의해 다른 코드가 실행되기 이전에 함수 객체가 먼저 생성된다. 그리고 **자바스크립트 엔진은 함수 이름과 동일한 이름의 식별자를 암묵적으로 선언하고 생성된 함수 객체를 할당한다.**

따라서 위 예제의 모든 함수는 자바스크립트 엔진에 의해 암묵적으로 선언된 함수 이름과 동일한 이름의 식별자에 할당된다. ①에서 함수 foo를 호출하면 자바스크립트 엔진은 함수를 호출하기 위해 먼저 함수를 가리키는 식별자 foo를 검색한다.

이처럼 함수도 식별자에 할당되기 때문에 스코프를 갖는다. 사실 함수는 식별자에 함수 객체가 할당된 것 외에는 일반 변수와 다를 바가 없다. 따라서 스코프를 "변수를 검색할 때 사용하는 규칙"이라고 표현하기보다는 "**식별자를 검색하는 규칙**"이라고 표현하는 것이 보다 적합하다.

4. 함수 레벨 스코프

'지역 스코프'를 함수로만 한정할 것인지(함수 레벨 스코프) vs. 모든 코드 블록으로 할 것인지

지역은 함수 몸체 내부를 말하고 지역은 지역 스코프를 만든다고 했다. 이는 **코드 블록이 아닌 함수에 의해서만 지역 스코프가 생성된다**는 의미이다.

C나 Java 등 대부분의 프로그래밍 언어는 함수 몸체 만이 아니라 모든 코드 블록(if, for, while, try/catch 등)이 지역 스코프를 만든다. 이러한 특성을 **블록 레벨 스코프**(Block level scope)라 한다. 하지만 var **키워드로 선언된 변수는 오로지 함수의 코드 블록 만을 지역 스코프로 인정한다.** 이러한 특성을 **함수 레벨 스코프**(Function level scope)라 한다. 아래 예제를 살펴보자.

JAVASCRIPT

```
var x = 1;

if (true) {
  // var 키워드로 선언된 변수는 함수의 코드 블록 만을 지역 스코프로 인정한다.
  // 함수 밖에서 선언된 변수는 코드 블록 내에서 선언되었다 할 지라도 모두 전역 변수이다.
  // 따라서 x는 전역 변수이다. 이미 선언된 전역 변수 x가 있으므로 변수 x는 중복 선언된다.
  // 이는 의도치 않게 변수값이 변경되는 부작용을 발생시킨다.
  var x = 10;
}

console.log(x); // 10
```

오타인 듯. 함수 레벨 스코프가 맞는 듯?

전역 변수 x가 선언되었고 if 문의 코드 블록 내에도 변수 x가 선언되었다. 이때 if 문의 코드 블록 내에서 선언된 변수 x는 전역 변수다. var 키워드로 선언된 변수는 블록 레벨 스코프를 인정하기 때문에 함수 밖에서 선언된 변수는 코드 블록 내에서 선언되었다 할 지라도 모두 전역 변수이다. 따라서 전역 변수 x는 중복 선언되고 그 결과 의도치 않은 전역 변수의 값이 재할당된다. 하나 더 예제를 살펴보자.

JAVASCRIPT

```
var i = 10;
```

```
// for문에서 선언한 i는 전역 변수이다. 이미 선언된 전역 변수 i가 있으므로 중복 선언된다.
for (var i = 0; i < 5; i++) {
  console.log(i); // 0 1 2 3 4
}

// 의도치 않게 변수의 값이 변경되었다.
console.log(i); // 5
```

결과값이 5가 나오는 이유는, 4까지 반환한 후, '5'로 한 번 더 평가 과정을 거침
->그런데 조건에 맞지 않는 false값->그 false값이 빠져나와서 콘솔 로그에 찍히는 것

블록 레벨 스코프를 지원하는 프로그래밍 언어에서는 for 문에서 반복을 위해 선언된 변수 i가 for 문의 코드 블록 내에서만 유효한 지역 변수이다. 이 변수를 for 문 외부에서 사용할 일은 없기 때문이다. 하지만 var 키워드로 선언된 변수는 블록 레벨 스코프를 인정하지 않기 때문에 변수 i는 전역 변수가 된다. 따라서 전역 변수 i는 중복 선언되고 그 결과 의도치 않은 전역 변수의 값이 재할당된다.

var 키워드로 선언된 변수는 오로지 함수의 코드 블록 만을 지역 스코프로 인정하지만, ES6에서 도입된 let, const 키워드는 블록 레벨 스코프를 지원한다. 이에 대해서는 "15. let, const와 블록 레벨 스코프"에서 살펴보도록 하자.

5. 렉시컬 스코프 그 유명한 클로저와 연결됨

아래 예제의 실행 결과를 예측해보자.

JAVASCRIPT

```
var x = 1;
```

x 에 undefined 할당 , foo라는 객체 할당, bar라는 객체 할당

```
function foo() {
  var x = 10;
  bar();
}
```

함수객체는 감춰진 프로퍼티에 자신의 상위 스코프를 갖고 있음->이게 클로저

```
function bar() {
  console.log(x);
}
```

x의 상위스코프가 어디일까? 가 문제

```
foo(); // ?
bar(); // ?
```

둘다 출력값 1 임

위 예제의 실행 결과는 함수 bar의 상위 스코프가 무엇인지에 따라 결정된다. 두가지 패턴을 예측할 수 있다.

1. **함수를 어디서 호출**했는지에 따라 함수의 상위 스코프를 결정한다.
2. **함수를 어디서 정의**했는지에 따라 함수의 상위 스코프를 결정한다.
   변수의 스코프 또한 변수가 선언된 곳임.

첫번째 방식으로 함수의 상위 스코프를 결정한다면 함수 bar의 상위 스코프는 함수 foo와 전역일 것이다. 두번째 방식으로 함수의 상위 스코프를 결정한다면 함수 bar의 상위 스코프는 전역일 것이다. 프로그래밍 언어는 일반적으로 이 두가지 방식 중 하나의 방식으로 함수의 상위 스코프를 결정한다.

첫번째 방식을 동적 스코프(Dynamic scope)라 한다. 함수 정의 시점에는 함수가 어디서 호출될 지 알 수 없다. 따라서 함수가 호출되는 시점에 동적으로 상위 스코프를 결정해야 하기 때문에 동적 스코프라고 부른다.

동적 스코프는 호출된 위치에 따라 상위 스코프가 항상 변하기 때문

두번째 방식을 렉시컬 스코프(Lexical scope) 또는 정적 스코프(Static scope)라 한다. 동적 스코프 방식처럼 상위 스코프가 동적으로 변하지 않고 함수 정의가 평가되는 시점에 상위 스코프가 정적으로 결정되기 때문에 정적 스코프라고 부른다. 자바스크립트를 비롯한 대부분의 프로그래밍 언어는 렉시컬 스코프를 따른다.

자바스크립트는 렉시컬 스코프를 따르므로 **함수를 어디서 호출했는지가 아니라 함수를 어디서 정의했는지에 따라 상위 스코프를 결정한다**. 즉, 모든 함수 정의(함수 선언문 또는 함수 표현식)는 평가되어 함수 객체를 생성할 때, 자신이 정의된 스코프를 상위 스코프로서 기억한다. 그리고 함수가 호출되면 언제나 자신이 기억하고 있는 자신이 정의된 스코프를 상위 스코프로 사용한다. 함수가 호출된 위치는 함수 자신이 기억하고 있는 스코프, 즉 상위 스코프 결정에 어떠한 영향을 주지 않는다.

위 예제의 bar 함수는 전역에서 정의된 함수이다. 함수 선언문으로 정의된 bar 함수는 전역 코드가 실행되기 전에 먼저 평가되어 함수 객체를 생성한다. 이때 생성된 bar 함수 객체는 자신이 정의된 스코프, 즉 전역 스코프를 기억한다. 그리고 bar 함수가 호출되면 호출된 곳이 어디인지 관계없이 언제나 자신이 기억하고 있는 전역 스코프를 상위 스코프로 사용한다. 따라서 위 예제를 실행하면 전역 변수 x의 값 1을 두 번 출력한다.

렉시컬 스코프는 클로저와 깊은 관계가 있다. 이에 대해서는 "24. 클로저"에서 자세히 살펴보도록 하자.