SCOPE

❓ 정의

> ⇒ 식별자(변수, 함수, 클래스)는 선언시 위치에 따라서 다른 코드 및 블럭에서 자신을 참조할 수 있는 유효범위가 설정된다. 이것을 스코프라 함

전역/지역 스코프

```
function add(x,y){
    console.log(x,y);
    return x + y;
}

add(2,5); // 함수는 정상적으로 동작

console.log(x,y); // x,y는 함수 내부에서 선언되어 있으므로 스코프가 맞지
```

⇒ 스코프는 블록 단위로 설정됨

```
var x = 'golbal'; // 전역 스코프
```

```
function foo(){
    var y = 'local'; // 지역 스코프
    console.log(y);
    console.log(x); // 함수 외부의 변수지만 전역 스코프라 참조가능
}

foo() // 정상동작

console.log(y) // 블럭안에 존재하는 지역 스코프라 참조불가 => 오류
```

⇒ 블록 내부의 지역 스코프는 해당 블록에서만 참조가능

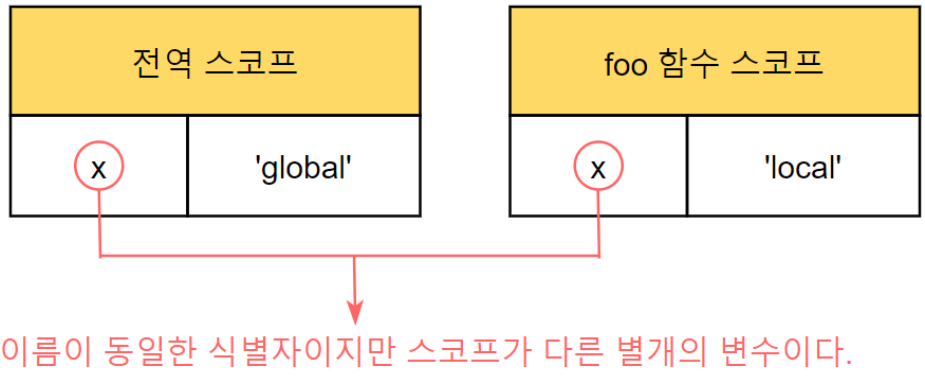
구분	설명	스코프	변수
전역	코드의 가장 바깥 영역	전역 스코프	전역 변수
지역	함수 몸체 내부	지역 스코프	지역 변수

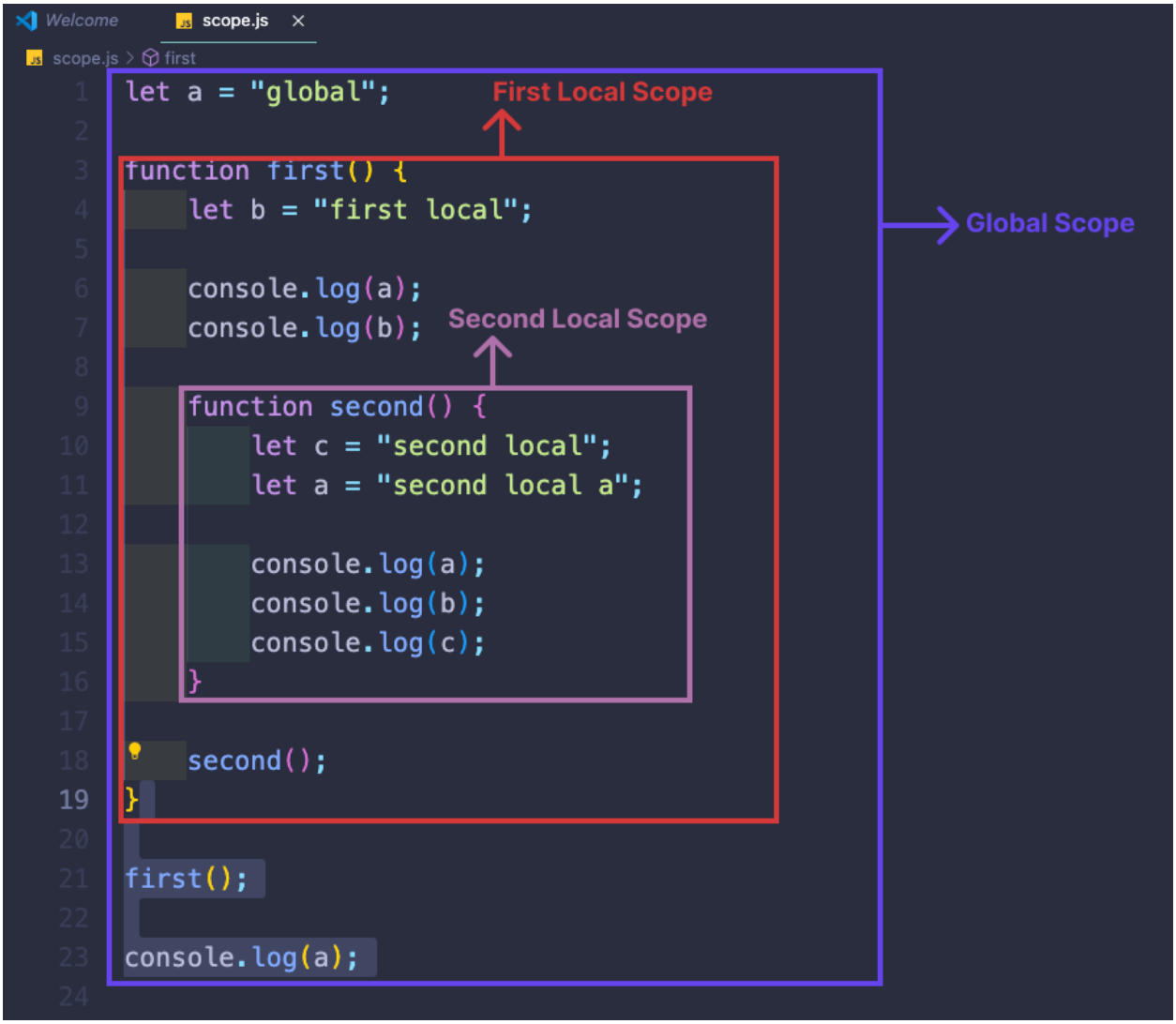
```
var x = 'golbal'; // 전역 스코프

function foo(){
    var x = 'local'; // 지역 스코프
    console.log(x);
}

foo() // ???
```

- 동일한 이름의 변수라 어떤걸 먼저 참조해야 할까?
- 답 : local 출력
- 지역 스코프 ⇒ 전역 스코프 순으로 확인
- Scope-Chain
- 스코프란 자바스크립트 엔진이 식별자(이름)을 참조하는 규칙
- x들은 동일한 식별자 but 다른 변수로 취급된다.





- 블록(함수) 단위로 스코프가 설정되는 모습

▼ 출력결과

1. global
2. first local
3. second local a
4. first local
5. second local
6. global

- 일종의 디렉토리 개념과 비슷
  - 같은 이름의 파일이라도 다른 폴더에 위치하면 구별가능

⇒ 즉, 스코프는 같은 이름이 존재 할 수 있는 존재 범위를 나타낸다.

⇒ 스코프는 자신(블록) 또는 하위 블록에서 유효함

!! var은 같은 스코프에 동일한 식별자로 선언이 된다. 이때 마지막 값으로 덮어씌워지므로 주의

```
function foo(){
    var a = 1;
    var a = 2;
    console.log(a); // 2 출력됨
}
```

- 이런 예측하지 못한 오류가 싫다면 중복 선언이 불가한 let, const 사용

스코프 체인

> ⇒ 블록(함수)의 중첩으로 인한 스코프의 계층적인 구조를 확인할 수 있다.

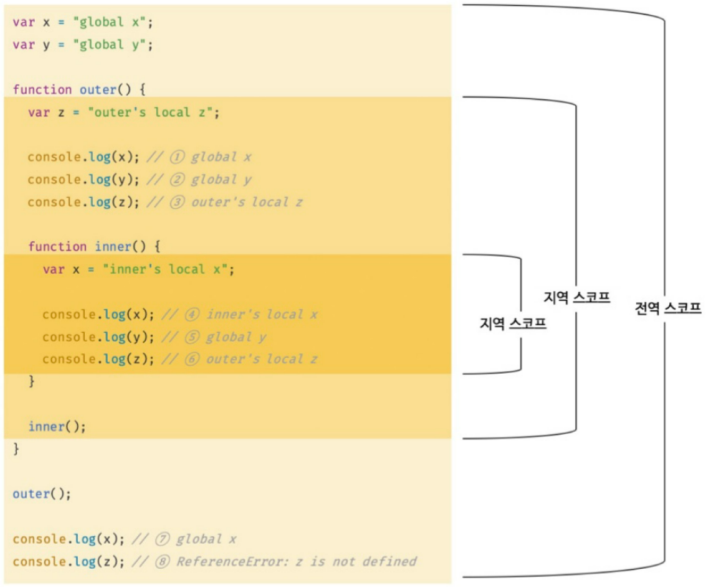
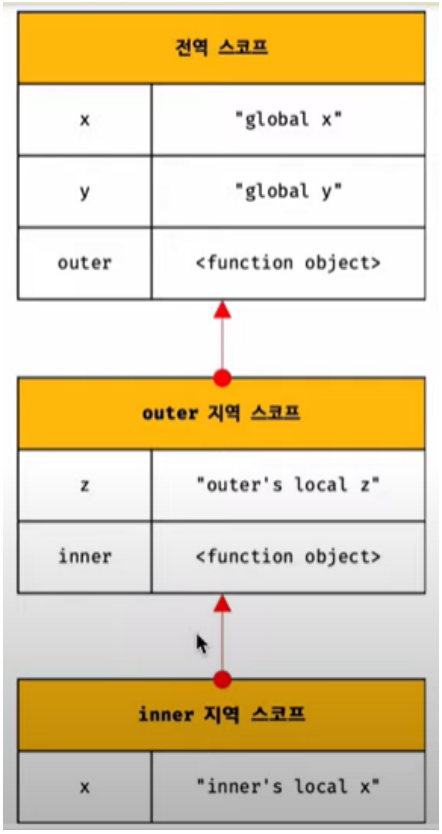


그림 13-2 전역 스코프와 지역 스코프



- 가장 안쪽의 중첩 블록의 지역 스코프부터 최상단의 전역 스코프까지 계층형태로 되어 있다.
  - ⇒ 스코프 체인
- 자바스크립트 엔진은 최하단부터 참조할 변수를 검색한다.(identifier resolution)
  - 상위 스코프의 변수를 하위 스코프에서 검색가능
  - 반대로 상위에서 하위 스코프는 검색 불가
  - 같은 식별자라도 하위의 식별자가 먼저 참조된다.

```
function foo(){
    console.log('global functio foo');
}
```

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

}

bar();
```

- 함수 또한 실행시 엔진이 동일한 이름의 식별자로 선언하여 함수 객체를 할당해 준다.
- 이로써 동일하게 변수처럼 사용할 수 있는 식별자로써 동작함
- 따라서 변수와 동일하게 식별자로 구분되어 스코프 체인이 동작한다.

함수 레벨 스코프

> ⇒ 스코프는 블록을 기준으로 나눠진다고 했었는데 엄밀히 구별하자면 2가지가 있다.

1. 블록 레벨 스코프
2. 함수 레벨 스코프

```
var x = 1;

function foo() {
    var x = 10;
    console.log(x);
}
```

```
foo();

console.log(x);
```

- 1 : 10 출력
- 2 : 1 출력
- 스코프 레벨 있음

```
var x = 1;

if (true) {
    var x = 10;
}

console.log(x);
////////////////////////////////////
var i = 10;

for (var i = 0; i < 5; i++) {
    console.log(i);
}

console.log(i);
// 출력 0,1,2,3,4,5
```

- 10 출력
- 블록 안쪽과 바깥쪽 동일한 스코프 레벨로 동작
- 블록 내부더라도 함수 바깥이라면 var 변수는 전부 전역변수로 동작
- for문의 조건문에서의 선언도 동일함

⇒ var의 경우 함수 레벨 스코프로 함수가 아닌 블록에서는 스코프레벨이 생기지 않는 것을 확인할 수 있다.

렉시컬 스코프

```
var x = 1;

function foo() {
    var x = 10; // 함수 스코프
    bar();
}

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

foo();
bar();
```

- 결과는 두가지를 예상할 수 있다.

1. 함수의 호출 위치(동적 스코프)
    a. 함수가 호출 되는 시점에서 스코프 레벨을 정함
    b. foo ← bar 순으로 레벨이 생김
    c. foo의 지역 스코프, 전역스코프가 bar의 상위 스코프
2. 함수의 선언 위치(렉시컬 스코프, 정적 스코프)
    a. 함수를 선언 하는 시점에 스코프 레벨을 정함
    b. 자바스크립트는 렉시컬 스코프를 사용함
    c. bar, foo는 동일한 스코프로 x = 1 을 출력함

▼ 조금 더 파보기

```
var x = 1;

function foo() {
    var x = 10;
    function bar() {
        console.log(x);
    }
    bar();
}

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

▼ 답

- foo() ← bar 순으로 레벨링이 되어 foo의 지역 스코프와 bar가 동일한 레벨이 되어 10출력
- bar()는 내부의 중첩 블럭으로 선언되었으므로 레벨이 맞지않아 오류

```
var x = 1;

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

foo(); // ?

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

bar(); // ?
```

▼ 답

- bar를 함수 표현식으로 선언 했기 때문에 변수 이름 자체는 호이스팅 되지만 함수는 할당되지 않아 undefined 형태
- 그래서 출력하려고 하면 오류

전역변수의 문제점

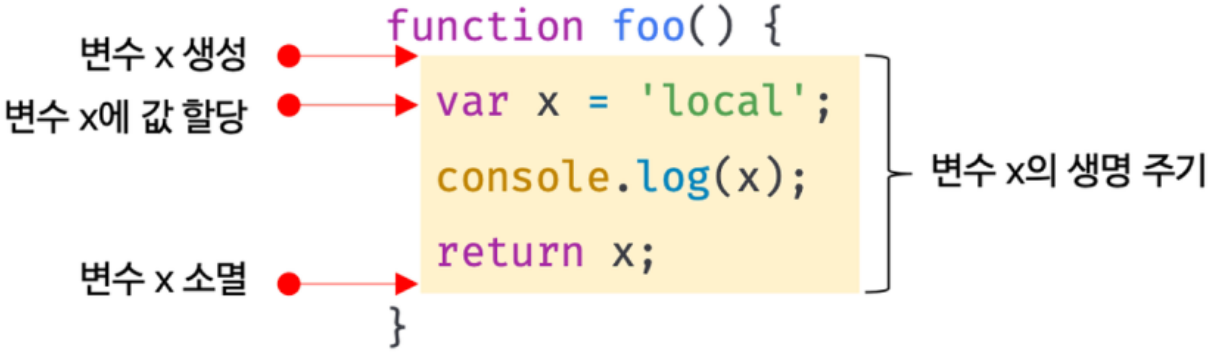
변수의 생명주기

> 메모리 공간의 확보(allocate) → 메모리 공간의 해제(release)까지의 메모리 풀에 공간이 반환되는 시점을 변수의 생명주기라 함

```
function foo(){
    var x = 'local';
    console.log(x);
    return x;
}

foo();
console.log(x);
```

- 외부의 console.log(x)에서 x가 존재(선언)하지 않으므로 오류




```
foo();
console.log(x);
```

- 호이스팅으로 인해 엔진이 런타임 이전에 변수를 선언 해준다. 단, 전역변수에만 해당
- 함수 내부의 변수의 경우 위의 생명주기를 지난다.
- 함수 시작시 변수 x생성, 하지만 데이터형은 undefined
- 이후 데이터형을 정하면서 할당
- 함수 종료시 변수도 소멸
  - 함수와 생명주기를 같이 가져간다.
  - 변수는 함수의 스코프에 등록되는 것
  - !! 만약 해당 스코프를 누군가 사용한다면 소멸되지 않음

```
var x = 'global';

function foo(){
    console.log(x); // <- 이 시점에 이미 x는 선언되어 있음 하지만 unde
    var x = 'local';
}
```

```
foo()
console.log(x);
```

- undefined, global 출력
- 호이스팅은 스코프 단위로 이루어짐

전역 변수의 생명주기

> 전역 변수의 경우 전역 객체의 프로퍼티로 존재, 전역 객체와 생명 주기를 동일하게 가져감

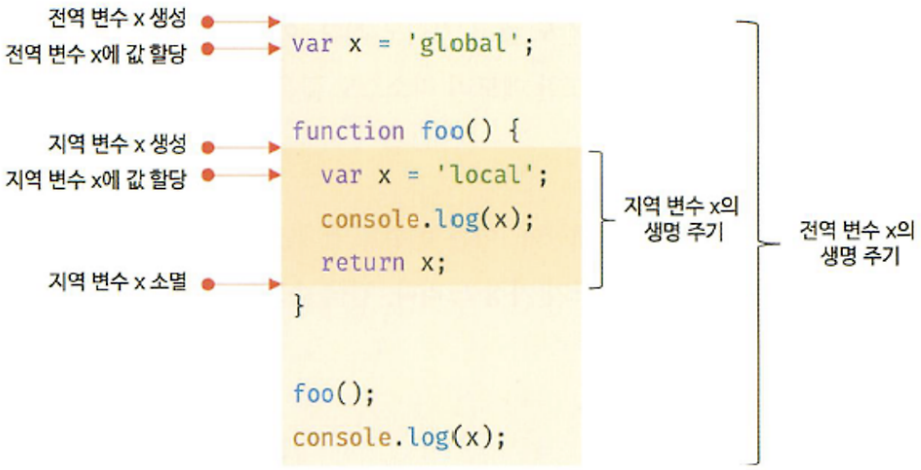


그림 14-2 전역 변수의 생명 주기

- 전역 객체 : globalThis
- 전역 객체는 엔진에 의해 다른 객체보다도 가장 먼저 생성됨

전역 변수의 문제점

1. 코드 어디서든 참조, 변경할 수 있는 암묵적 결함을 허용함
    - 이것은 변수의 유효 범위가 커져 코드의 가독성 하락과 의도치 않게 나쁜 상태를 유도할 수 있음
2. 긴 생명주기
    - 생명주기가 길어 메모리 리소스를 오래 잡고 있음
    - var의 경우 코드상 이름이 중복될 수 있어 의도치 않은 참조로 인한 오류가 발생할 수 있음
3. 스코프 체인 상에서 종점에 존재
    - 가장 상단의 스코프 레벨에 위치하므로 검색시 다른 하위 스코프 레벨과 비교하여 속도가 느린 단점이 있다.
4. 네임스페이스 오염
    - 자바스크립트의 고유한 특징으로 다른 파일에 있더라도 전역변수끼리 전역 스코프를 공유하므로 의도치 않으 오류를 일으킬 가능성이 있다.

해결책

1. 즉시 실행 함수(IIFE, Immediately Invoked Function Expression)

```
(function () {
    var foo = 10;
    // ...
})();

console.log(foo);
```

    - 이렇게 하면 전역 변수가 생기지 않는다.
    - **괄호의 의미**
        - **함수 표현식**으로 인식하게 만들어서 익명 함수가 정의될 수 있도록 하기 위함

- 괄호를 씌우지 않으면 자바스크립트는 이를 함수 선언으로 간주하고 이름이 없는 함수 선언은 문법적으로 오류를 일으킴

```
var myFunction = function() {
    console.log('This is a function expression');
}();
```

2. 네임스페이스 객체
   - 전역 네임스페이스 객체를 이용한 전역 변수선언

```
var MYAPP = {};

MayAPP.person = {
    name:'Lee';
    address: 'Seoul';
}

console.log(MYAPP.person.name);
```

   - 객체를 통해서 변수를 선언하므로 식별자 충돌방지에 용이
   - 하지만 결국 객체 그 자체도 전역변수이므로 주의 필요
3. 모듈 패턴
4. ES6 모듈

let, const 키워드와 블록레벨 스코프

var 키워드

```
var x = 1;
var y = 1;
```

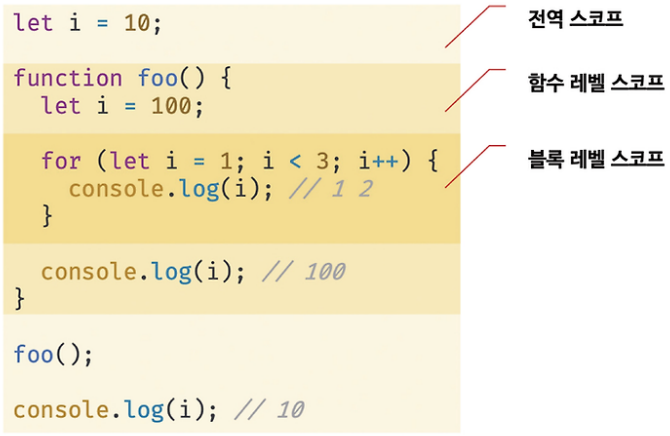
```
var x = 100;
var y;

console.log(x); // 100 출력
console.log(y); // 1 출력
```

- var의 경우 중복 선언이 허용됨
- 초기화 하지 않는 변수는 무시됨

let 키워드

- 중복 선언이 허용되지 않음
- 블록 레벨 스코프



변수 호이스팅

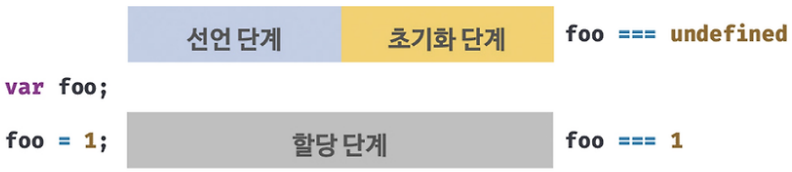
```
console.log(foo); // undefined 출력

foo = 123;
```

```
console.log(foo); // 123 출력

var foo;
```

- 변수 호이스팅에 의해 첫번째 줄 동작
- 하지만 foo가 아직 할당되지 않았으므로 undefined 출력

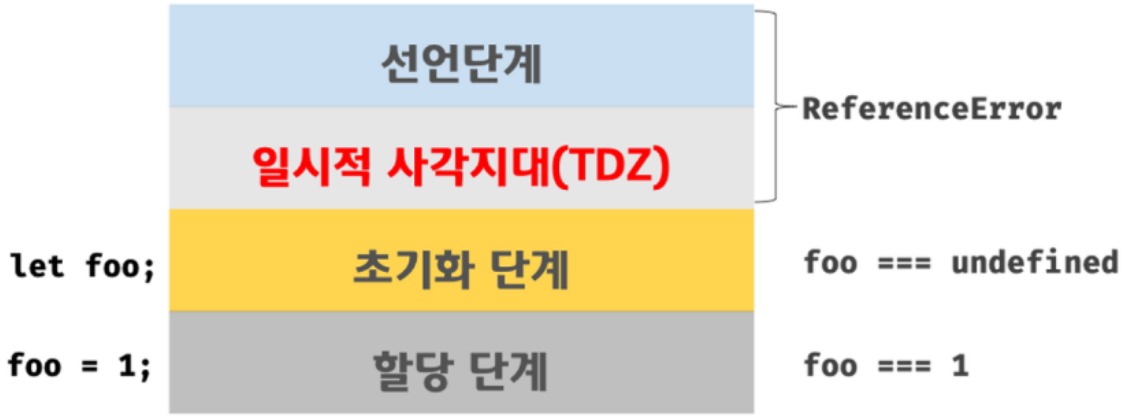


- 런타임이럿 엔진에서 선언, 초기화가 이루어짐

```
console.log(foo); //초기화 이전 접근 => 오류

let foo;
console.log(foo); // undefined

foo = 1;
console.log(foo); // 1
```



- 엔진이 런타임 이전에 선언해 줌

- 단, let는 선언와 초기화가 분리되어 있음
- 만약 초기화전 접근시 오류
- 변수 선언시 초기화 진행
    - 변수 선언까지 와야 에러가 나지 않아 호이스팅이 이루어지지 않는 것처럼 보이지만 실제로는 일어난다.

```
let foo = 1;

{
    console.log(foo); // error
    let foo = 2;
}
```

- 호이스티잉 일어나지 않는다면 상위 스코프의 foo가 있으므로 동작해야 하지만 오류가 남
    - 블록 내부의 foo가 존재하여 호이스팅이 일어남
    - 아직 초기화가 이루어지지 않아 오류가 남

```
// 이 예제는 브라우저 환경에서 실행해야 한다.

// 전역 변수
var x = 1;
// 암묵적 전역
y = 2;
// 전역 함수
function foo() {}

// var 키워드로 선언한 전역 변수는 전역 객체 window의 프로퍼티다.
console.log(window.x); // 1
// 전역 객체 window의 프로퍼티는 전역 변수처럼 사용할 수 있다.
console.log(x); // 1
```

```
// 암묵적 전역은 전역 객체 window의 프로퍼티다.
console.log(window.y); // 2
console.log(y); // 2

// 함수 선언문으로 정의한 전역 함수는 전역 객체 window의 프로퍼티다.
console.log(window.foo); // ƒ foo() {}
// 전역 객체 window의 프로퍼티는 전역 변수처럼 사용할 수 있다.
console.log(foo); // ƒ foo() {}
```

- var의 경우 window 객체에 할당되어 다음과 같이 동작할 수 있음
- !! Node.JS
  - node의 경우 모듈로 동작하므로 globalThis, window 객체에 할당되지 않아 window, globalThis 객체에 직접 할당해야 함

```
    // 이 예제는 브라우저 환경에서 실행해야 한다.

    let x = 1;

    // let, const 키워드로 선언한 전역 변수는 전역 객체 window의 프로퍼티

    console.log(window.x); // undefined

    console.log(x); // 1
```

- let의 경우 window 객체에 할당되지 않음
- const도 동일

const 키워드

- let와 대부분 동일하지만 선언시 초기화까지 해줘야 하는 다른 점이 있다.
- 값을 변경할 수 없는 특징이 있다.

- 단, 객체를 할당한 경우 내부의 값을 변경할 수 있다.

```
cost person = {
    name: 'Lee'
};

person.name = 'Kim';

console.log(person); // {name: 'Kim'}
```