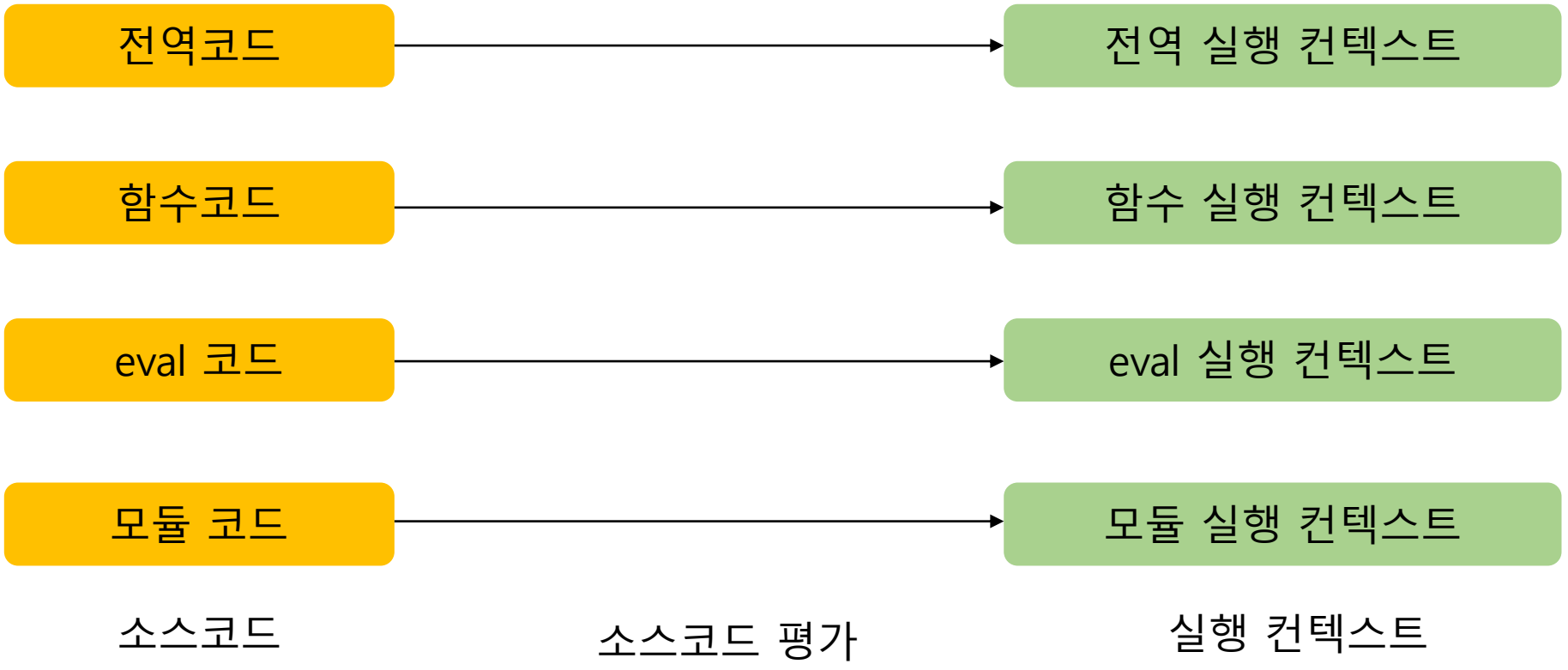
# 모던 자바스크립트
# Deep Dive

# 23장_실행 컨텍스트

# 소스코드 타입

ECMA Script 사양은 소스코드를 **4가지 타입**으로 구분
구분하는 이유는, 소스코드의 타입에 따라 실행 컨텍스트 생성하는 과정과 관리 내용이 다름



# 소스코드 타입

**1. 전역 코드**
전역에 존재하는 소스코드. 전역에 정의된 함수, 클래스 내부코드 포함X
전역 코드는 전역 변수를 관리하기 위해 최상위 스코프인 전역 스코프를 생성해야 한다. 그리고 var 키워드로 선언된 전역 변수와 함수 선언문으로 정의된 전역 함수를 전역 객체의 프로퍼티와 메서드로 바인딩하고 참조하기 위해 전역 객체와 연결되어야 한다. 이를 위해 전역 코드가 평가되면 전역 실행 컨텍스트가 생성된다.

**2. 함수 코드**
함수 내부에 존재하는 소스코드. 함수 내부에 중첩된 함수, 클래스 내부코드 포함X
함수 코드는 지역 스코프를 생성하고 지역 변수, 매개 변수, arguments 객체를 관리해야 한다. 그리고 생성한 지역 스코프를 전역 스코프에서 시작하는 스코프 체인의 일원으로 연결해야 한다. 이를 위해 함수 코드가 평가되면 함수 실행 컨텍스트가 생성된다.

**3. eval 코드**
eval 함수에 인수로 전달되는 소스코드
strict mode에서 독자적인 스코프 생성. 이를 위해 함수 코드가 평가되면 eval 실행 컨텍스트 생성

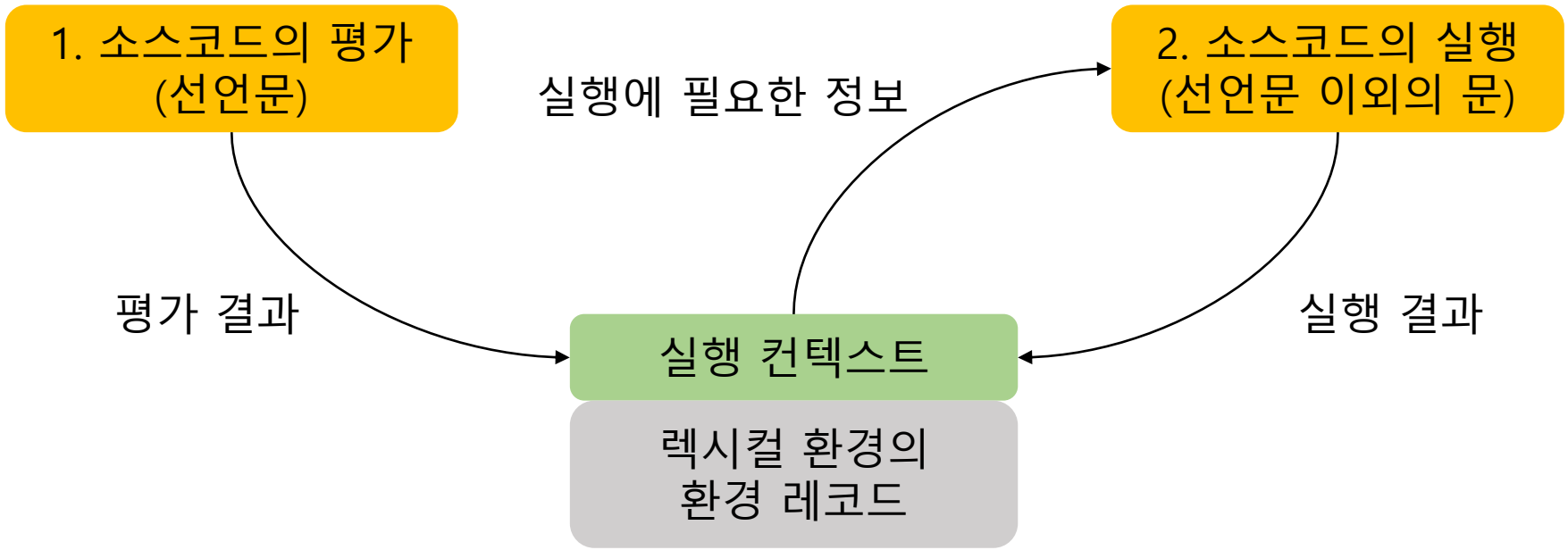
**4. 모듈 코드**
모듈 내부에 존재하는 소스코드. 모듈 내부의 함수, 클래스 등의 내부코드 포함X
모듈 코드는 모듈별로 독립적인 모듈 스코프 생성. 이를 위해 모듈 코드가 평가되면 모듈 실행 컨텍스트 생성

# 소스코드 평가와 실행

자바스크립트 엔진은 소스코드를 두가지 과정으로 나누어서 실행
자바스크립트 엔진은 하는 일이 딱 두가지 (1) 평가 (2) 실행

**(1) 평가(런타임 이전)**
실행 컨텍스트 생성
변수, 함수 선언문 식별자는 실행 컨텍스트가 관리하는 렉시컬 환경의 환경 레코드에 등록

**(2) 실행(런타임)**
실행에 필요한 정보는 렉시컬 환경의 환경 레코드에 검색해서 취득
소스코드 실행 결과(참조 값의 변경 등)는 렉시컬 환경의 환경 레코드에 등록



# 소스코드 평가와 실행

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

**소스코드의 평가**

| 실행 컨텍스트 | |
|---|---|
| x | undefined |

**소스코드 실행**

| 실행 컨텍스트 | |
|---|---|
| x | 1 |

# 실행 컨텍스트의 역할

**자바스크립트 엔진이 이 예제를 어떻게 평가하고 실행하는지 생각해보자**

**1. 전역 코드 평가**
변수 선언문, 함수 선언문이 먼저 실행
전역 변수와 전역 함수가 실행 컨텍스트의 전역 스코프에 등록
var 키워드로 선언된 전역 변수, 함수 선언문으로 정의된 전역 함수는 전역 객체에 등록

**2. 전역 코드 실행**
전역 코드가 순사적으로 실행
전역 변수에 값이 할당되고 함수가 호출
함수가 호출되면 전역 코드의 실행을 일시 중단하고, 함수 내부로 진입

**3. 함수 코드 평가**
함수 내부로 진입 후, 함수 코드 평가 과정을 거침
매개 변수와 지역 변수 선언문이 먼저 실행
매개 변수와 지역 변수가 실행 컨텍스트의 지역 스코프에 등록

```
1  // 전역 변수 선언
2  const x = 1;
3  const y = 2;
4
5  // 함수 정의
6  function foo(a) {
7    // 지역 변수 선언
8    const x = 10;
9    const y = 20;
10
11   // 메서드 호출
12   console.log(a + x + y); // 130
13 }
14
15 // 함수 호출
16 foo(100);
17
18 // 메서드 호출
19 console.log(x + y); // 3
```

# 실행 컨텍스트의 역할

**4. 함수 코드 실행**

함수 코드 순차적으로 실행

매개 변수와 지역 변수에 값 할당 및 console.log 호출

(1) console.log를 호출하기 위해서 console을 스코프 체인을 통해서 검색

(2) console 식별자는 전역 객체에 프로퍼티로 존재

(3) log 프로퍼티를 console객체의 프로토타입 체인을 통해 검색

(4) 이후 a, x, y를 스코프 체인을 통해 검색 후 a + x + y가 평가

(5) console.log 메서드 실행이 종료되면 함수 호출 이전으로 되돌아가 전역 코드 실행

```
1  // 전역 변수 선언
2  const x = 1;
3  const y = 2;
4
5  // 함수 정의
6  function foo(a) {
7    // 지역 변수 선언
8    const x = 10;
9    const y = 20;
10
11   // 메서드 호출
12   console.log(a + x + y); // 130
13 }
14
15 // 함수 호출
16 foo(100);
17
18 // 메서드 호출
19 console.log(x + y); // 3
```

# 실행 컨텍스트란?

코드가 실행되려면 식별자, 스코프, 코드 실행 순서 관리 필요

1. 선언된 모든 식별자는 스코프를 구현하여 등록하고 상태 변화 지속적으로 관리
2. 스코프는 중첩 관계에 의해 스코프 체인 형성 및 스코프 체인을 통해서 식별자를 검색
3. 현재 실행 중인 코드의 실행 순서 변경(함수 호출) 및 다시 되돌아갈 수도 있어야 함

실행 컨텍스트란?

1. 실행 컨텍스트는 소스코드를 실행하는 데 필요한 환경을 제공하고 코드의 실행 결과를 실제로 관리하는 영역
2. 실행 컨텍스트는 식별자를 등록하고 관리하는 스코프와, 코드 실행 순서 관리를 구현한 내부 메커니즘으로, 모든 코드는 실행 컨텍스틀 통해 실행되고 관리된다.
3. 식별자와 스코프는 렉시컬 환경을 통해서, 코드 실행 순서는 실행 컨텍스트 스택으로 관리

실행 컨텍스트는 **렉시컬 환경**으로 **식별자, 스코프**를, **실행 컨텍스트 스택**으로 **코드 실행 순서** 관리

# 실행 컨텍스트 스택

자바스크립트 엔진은,
**전역 코드**를 평가하여 **전역 실행 컨텍스트 생성**
**함수 코드**를 평가하여 **함수 실행 컨텍스트 생성**

이때 생성된 실행 컨텍스트는 스택 자료구조로 관리. 이를 **실행 컨텍스트 스택**이라고 함
코드를 실행하면, 시간의 흐름에 따라 실행 컨텍스트 스택에 실행 컨텍스트가 push되고 pop됨

```
1   const x = 1;
2
3   function foo() {
4       const y = 2;
5       function bar() {
6           const z = 3;
7           console.log(x + y + z);
8       }
9       bar();
10  }
11  foo();
```

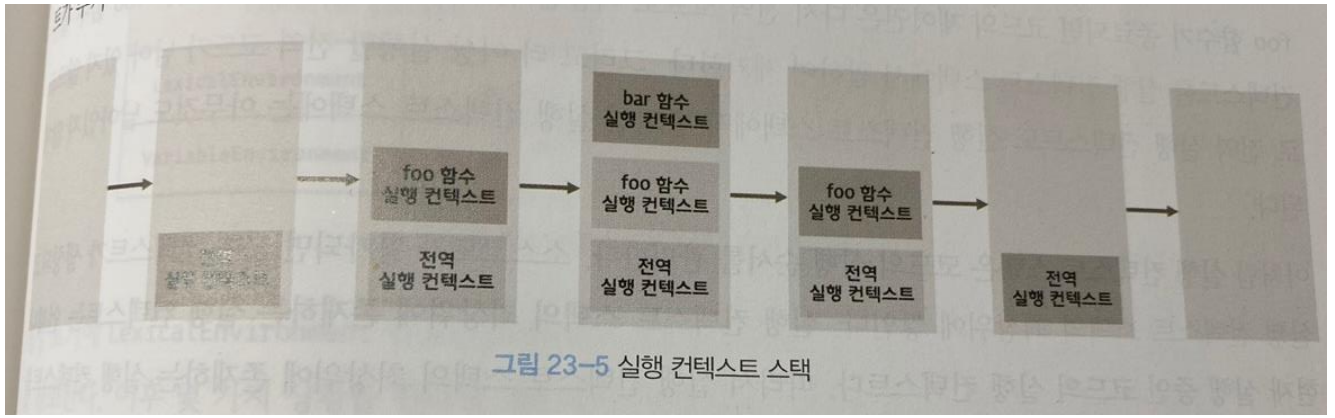


그림 23-5 실행 컨텍스트 스택

# 실행 컨텍스트 스택

**1. 전역 코드의 평가와 실행**
자바스크립트 엔진은 전역 코드를 평가하여 실행 컨텍스트 생성 및 실행 컨텍스트 스택에 푸시
이때 전역 변수 x와 전역 함수 foo는 전역 실행 컨텍스트에 등록
전역 코드가 실행되면 값 할당 및 전역 함수 foo 호출

**2. foo 함수 코드의 평가와 실행**
전역 함수 foo가 호출되면 전역 코드의 실행은 일시 중단되고 **코드의 제어권이 foo함수 내부로 이동**
자바스크립트 엔진은 함수 코드를 평가하여 foo 함수 실행 컨텍스트 생성 및 실행 컨텍스트 스택에 푸시
foo함수의 지역 변수 y와 중첩 함수 bar가 foo 함수 실행 컨텍스트에 등록
foo 함수 코드가 실행되기 시작하여 지역 변수 y에 값이 할당 및 중첩 함수 bar호출

```
1   const x = 1;
2
3   function foo() {
4       const y = 2;
5       function bar() {
6           const z = 3;
7           console.log(x + y + z);
8       }
9       bar();
10  }
11  foo();
```

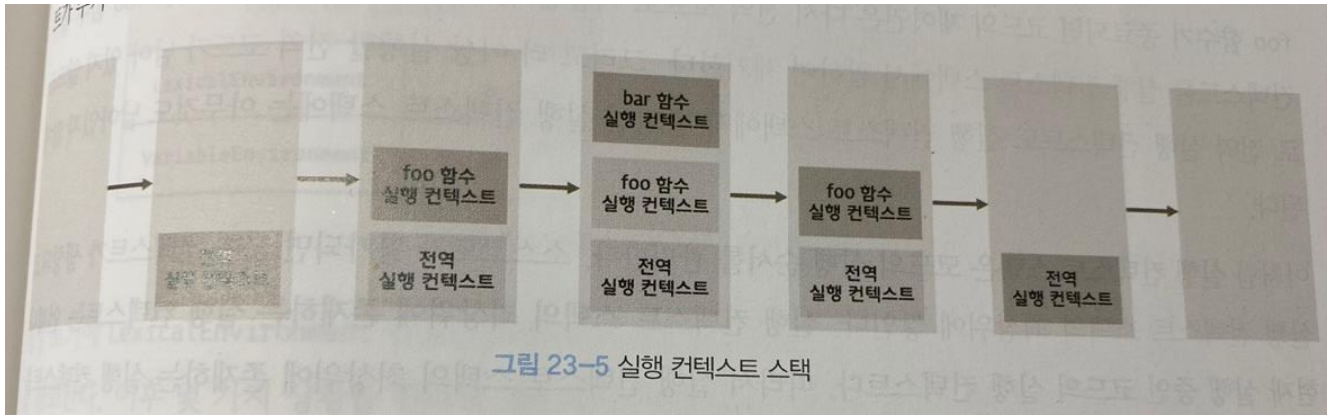


그림 23-5 실행 컨텍스트 스택

# 실행 컨텍스트 스택

**3. bar 함수 코드의 평가와 실행**

중첩 함수 bar가 호출되면 foo 함수 코드의 실행은 일시 중단되고 코드의 제어권이 bar 함수 내부로 이동
자바스크립트 엔진은 함수 코드를 평가하여 bar 함수 실행 컨텍스트 생성 및 실행 컨텍스트 스택에 푸시
bar 함수의 지역 변수 z가 함수 실행 컨텍스트에 등록
bar 함수 코드가 실행되기 시작하여 z에 값이 할당되고, console.log 메서드 호출 후 bar 함수 종료
(console.log 메서드도 함수이므로, 호출되면 실행 컨텍스트 생성하고 실행 컨텍스트 스택에 푸시되나 여기서는 생략)

**4. foo 함수 코드로 복귀**

bar 함수가 종료되면 코드의 제어권은 다시 foo 함수로 이동
자바스크립트 엔진은 bar 함수를 실행 컨텍스트 스택에서 pop
foo 함수는 더이상 실행할 코드가 없으므로 종료

```
1   const x = 1;
2
3   function foo() {
4       const y = 2;
5       function bar() {
6           const z = 3;
7           console.log(x + y + z);
8       }
9       bar();
10  }
11  foo();
```

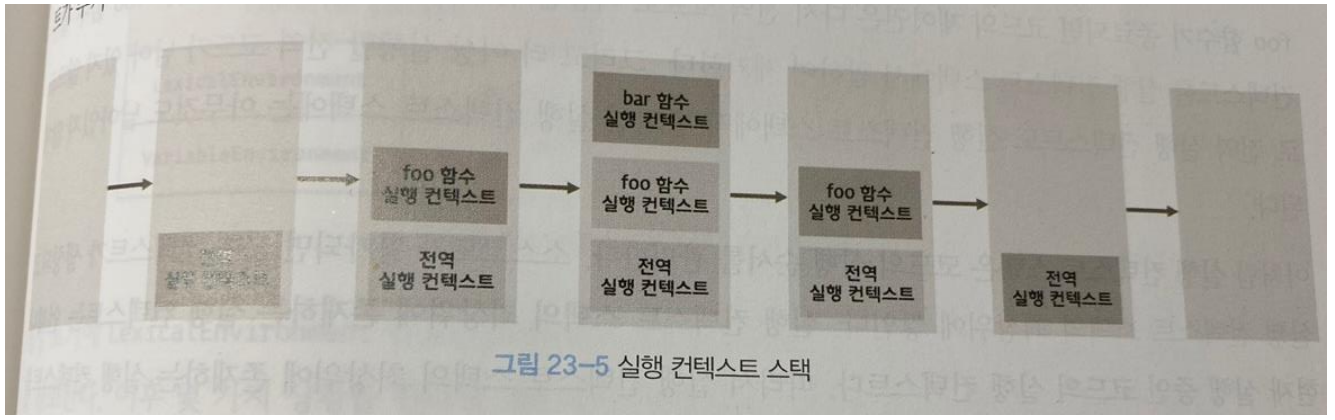


그림 23-5 실행 컨텍스트 스택

# 실행 컨텍스트 스택

**5. 전역 코드로의 복귀**
foo 함수가 종료되면 코드의 제어권은 다시 전역 코드로 이동.
자바스크립트 엔진은 foo 함수 실행 컨텍스트를 실행 컨텍스트 스택에서 pop하여 제거
더이상 실행할 전역 코드가 남아있지 않으므로, 전역 실행 컨텍스트도 실행 컨텍스트 스택에서 pop
실행 컨텍스트 스택에는 아무것도 남아있지 않게됨

실행 컨텍스트는 코드의 실행 순서를 관리
실행 컨텍스트 스택의 최상위에 존재하는 실행 컨텍스트는 현재 실행 중인 코드의 실행 컨텍스트
이를 실행 중인 실행 컨텍스트(running execution context)라 부름

```
1   const x = 1;
2
3   function foo() {
4       const y = 2;
5       function bar() {
6           const z = 3;
7           console.log(x + y + z);
8       }
9       bar();
10  }
11  foo();
```

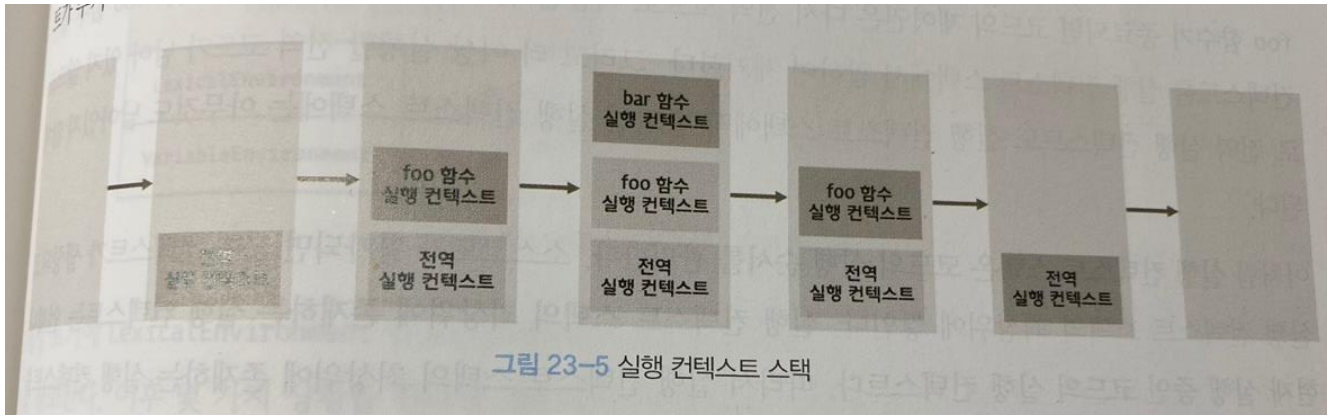


그림 23-5 실행 컨텍스트 스택

# 렉시컬 환경

실행 컨텍스트를 구성하는 컴포넌트
**(1)식별자와 식별자에 바인딩된 값**, **(2)상위 스코프에 대한 참조를 기록**하는 **자료구조**
키와 값을 갖는 객체 형태의 스코프에 식별자를 키로하여 식별자에 바인딩된 값을 관리
렉시컬 환경은 **스코프의 실체**

실행 컨텍스트는 **(1)렉시컬 환경 컴포넌트**와 **(2)가변 환경 컴포넌트**로 구성

생성 초기에 렉시컬 환경 컴포넌트와 가변 환경 컴포넌트는 동일한 렉시컬 환경 참조. 이후 몇 가지 상황을 만나면 가변 환경 컴포넌트를 위한 새로운 렉시컬 환경을 생성하고, 이때부터 렉시컬 환경 컴포넌트와 가변 환경 컴포넌트는 내용이 달라지는 경우가 존재. 이 책에서는 strict mode와 eval 코드, try/catch문과 같은 특수한 상황은 제외하고,

렉시컬 환경 컴포넌트와 가변 환경 컴포넌트도 구분하지 않고 **렉시컬 환경으로 통일해서 간략하게 설명**

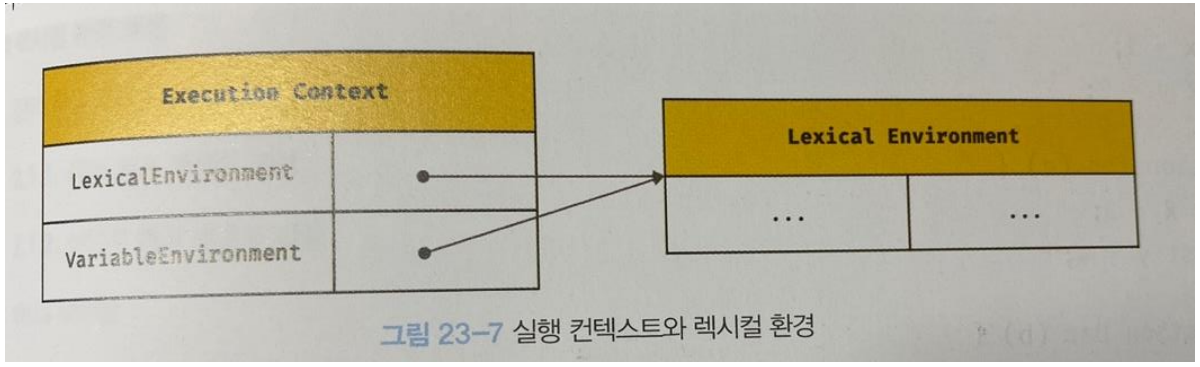


그림 23–7 실행 컨텍스트와 렉시컬 환경

# 실행 컨텍스트 생성 및 식별자 검색 과정

다음 코드가 어떻게 평가되고 실행되는지 알아보자.

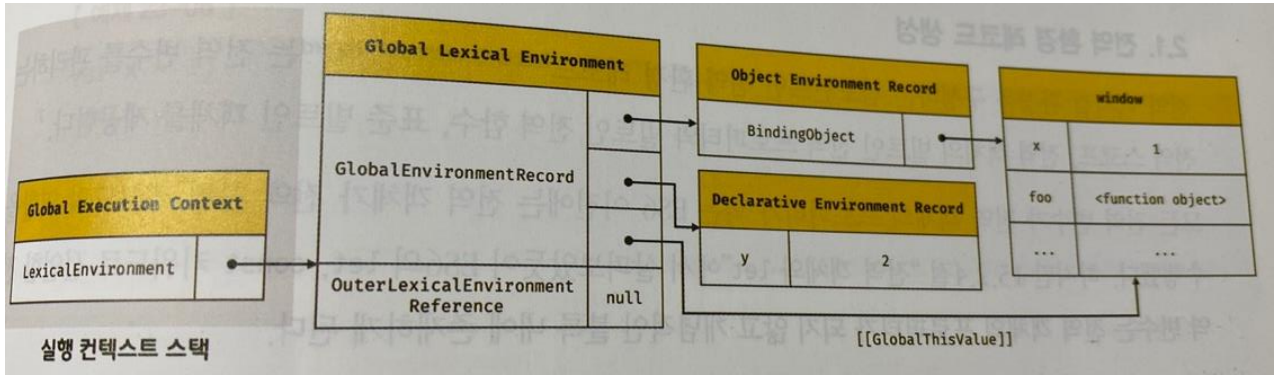
**1. 전역 객체 생성**
**2. 전역 코드 평가**
**3. 전역 코드 실행**
**4. foo 함수 코드 평가**
**5. foo 함수 코드 실행**
**6. bar 함수 코드 평가**
**7. bar 함수 코드 실행**

```
1   var x = 1;
2   const y = 2;
3
4   function foo(a) {
5     var x = 3;
6     const y = 4;
7
8     function bar(b) {
9       const z = 5;
10      console.log(a + b + x + y + z);
11    }
12
13    bar(10);
14  }
15
16  foo(20); // 42
```

# 실행 컨텍스트 생성 및 식별자 검색 과정

**1. 전역 객체 생성**
전역 코드 **평가 이전에 생성**
전역 객체에는 (1) 빌트인 전역 프로퍼티와 메서드 (2) 표준 빌트인 객체 (3) 호스트 객체 등록
전역 객체도 Object.prototype을 상속받으며, 프로토타입 체인의 일원

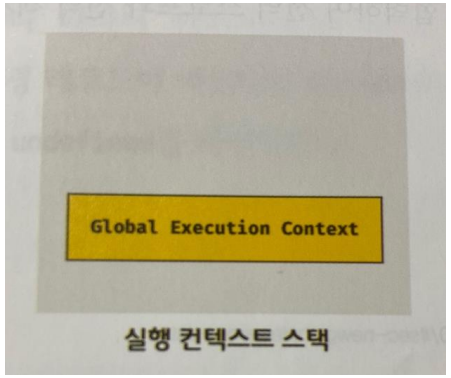
**2. 전역 코드 평가 및 실행**
2.1. 전역 실행 컨텍스트 생성
2.2. 전역 렉시컬 환경 생성
2.2.1. 전역 환경 레코드 생성
2.2.2.1. 객체 환경 레코드 생성
2.2.2.2. 선언적 환경 레코드 생성
2.2.2. this 바인딩
2.2.3. 외부 렉시컬 환경 참조 결정



# 실행 컨텍스트 생성 및 식별자 검색 과정

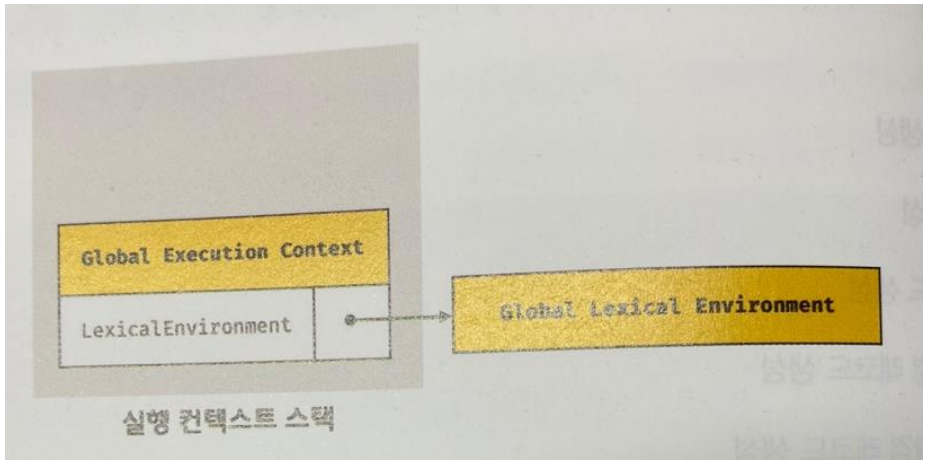
**2.1. 전역 실행 컨텍스트 생성**

실행 컨텍스트 스택에 비어있는 전역 실행 컨텍스트 생성 및 push
실행 컨텍스트 스택의 최상위에 있으므로, 현재 실행 중인 실행 컨텍스트가 됨




**2.2. 전역 렉시컬 환경 생성**

전역 렉시컬 환경 생성 및 전역 실행 컨텍스트에 바인딩
**전역 렉시컬 환경**은 (1)**전역 환경 레코드**와 (2)**외부 렉시컬 환경에 대한 참조** 컴포넌트로 구성
**전역 환경 레코드**는 (1)**객체 환경 레코드**와 (2)**선언적 환경 레코드**로 구성. + **내부 슬롯 [[globalThisValue]]** 존재



# 실행 컨텍스트 생성 및 식별자 검색 과정

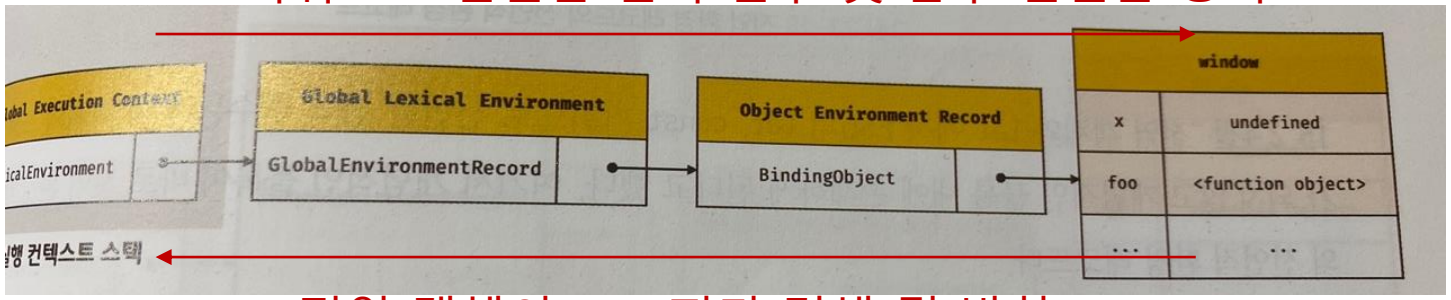
**2.2.1 객체 환경 레코드**

객체 환경 레코드는 **BindingObject객체(전역객체)**와 연결

var 키워드로 선언한 전역 변수, 함수 선언문으로 정의된 전역 함수는 전역 환경 레코드의 객체 환경 레코드에 연결된 Binding Object를 통해 전역 객체의 프로퍼티와 메서드가 됨

식별자를 전역 환경 레코드의 객체 환경 레코드에서 검색하면, 전역 객체의 프로퍼티를 검색 및 반환

var 키워드로 선언한 전역 변수 및 함수 선언문 등록




전역 객체의 프로퍼티 검색 및 반환

# 실행 컨텍스트 생성 및 식별자 검색 과정

**2.2.2 선언적 환경 레코드**

let, const 키워드로 선언한 전역 변수는 선언적 환경 레코드에 등록 및 관리
(var 키워드로 선언한 전역 변수, 함수 선언문으로 정의된 전역 함수 이외의 선언)
window.y로 접근이 불가능

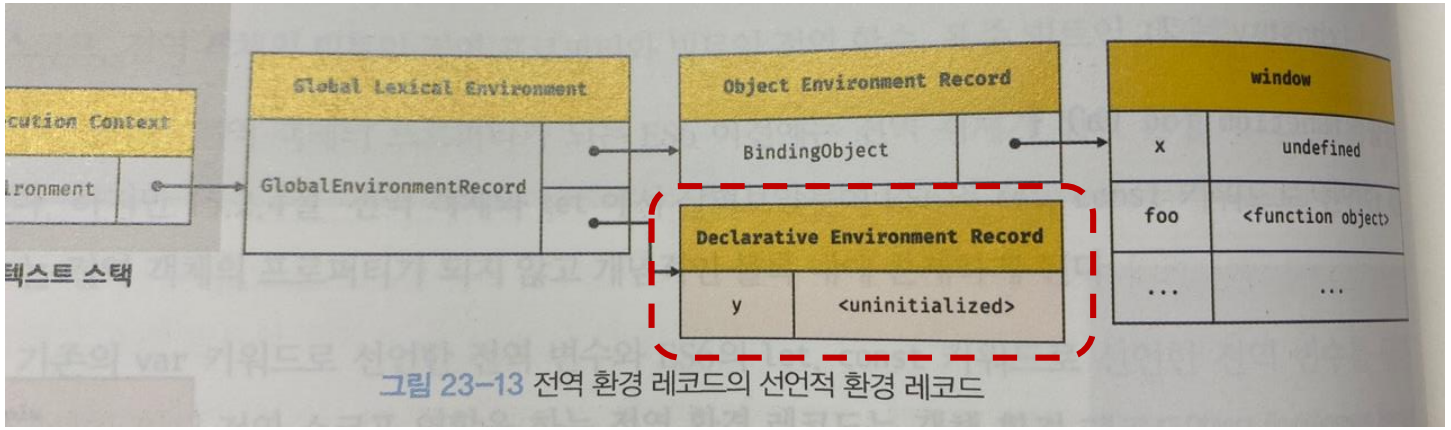


그림 23-13 전역 환경 레코드의 선언적 환경 레코드

y는 const로 선언하였으므로, TDZ에 빠지기 때문에, <uninitialized>로 기술
실제 <uninitialized>값이 바인딩 된 것은 아님

# 실행 컨텍스트 생성 및 식별자 검색 과정

### 2.2.3 [[globalThisValue]] 내부 슬롯

[[globalThisValue]] 내부 슬롯에는 **this가 바인딩**

전역 코드에서 this는 전역 객체를 가리키므로, [[globalThisValue]] 내부 슬롯은 전역 객체 바인딩

[[globalThisValue]]는 전역 렉시컬 환경과 함수 렉시컬 환경에만 존재

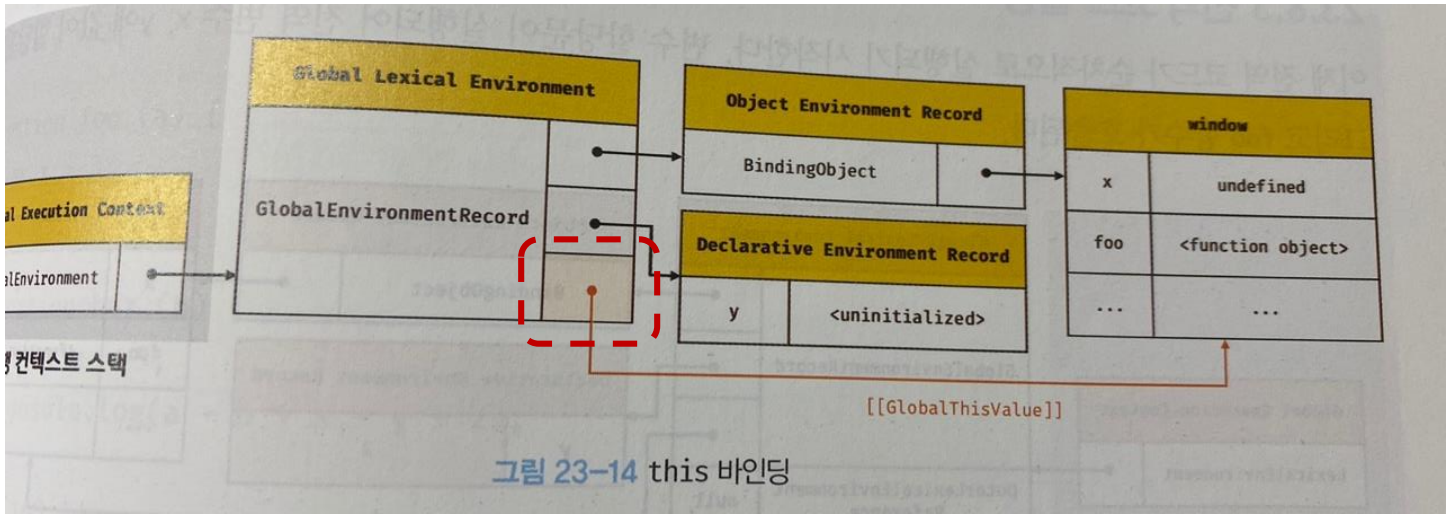



그림 23-14 this 바인딩

# 실행 컨텍스트 생성 및 식별자 검색 과정

**2.3 외부 렉시컬 환경에 대한 참조(Outer Lexical Environment Reference)**
현재 평가중인 소스코드를 포함하는 외부 소스코드의 렉시컬 환경, 즉 **상위 스코프 참조**
이를 통해서 **단방향 링크드 리스트** 구현

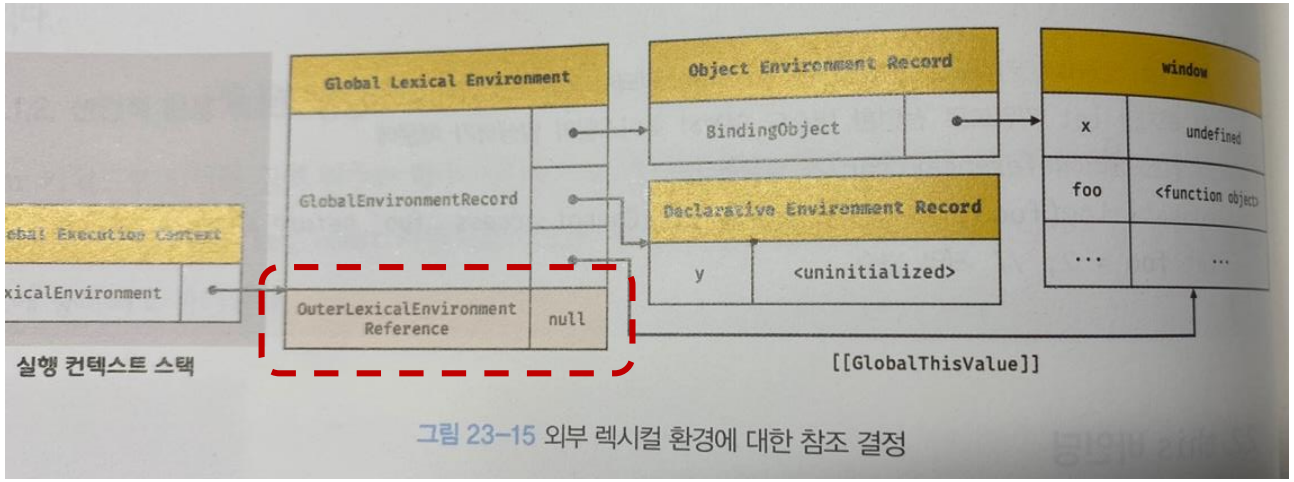


그림 23-15 외부 렉시컬 환경에 대한 참조 결정

전역 코드의 전역 렉시컬 환경은 스코프 체인의 종점에 존재하므로, null을 가리킴

# 실행 컨텍스트 생성 및 식별자 검색 과정

**3. 전역 코드 실행**
변수 할당문과 함수 호출문을 실행하려면, 선언된 식별자인지 확인.
식별자는 스코프가 다르면 같은 이름을 가질 수 있음. 어느 스코프의 식별자를 참조할지 **'식별자 결정'**을 해야함

**스코프 체인의 동작 원리**
식별자 결정은 실행 중인 실행 컨텍스트의 렉시컬 환경부터 검색하기 시작
만약 검색할 수 없으면 **외부 렉시컬 환경에 대한 참조가 가리키는 렉시컬 환경(상위 스코프)으로 이동**하여 검색
검색할 수 없는 경우, 참조 에러(Reference Error) 발생

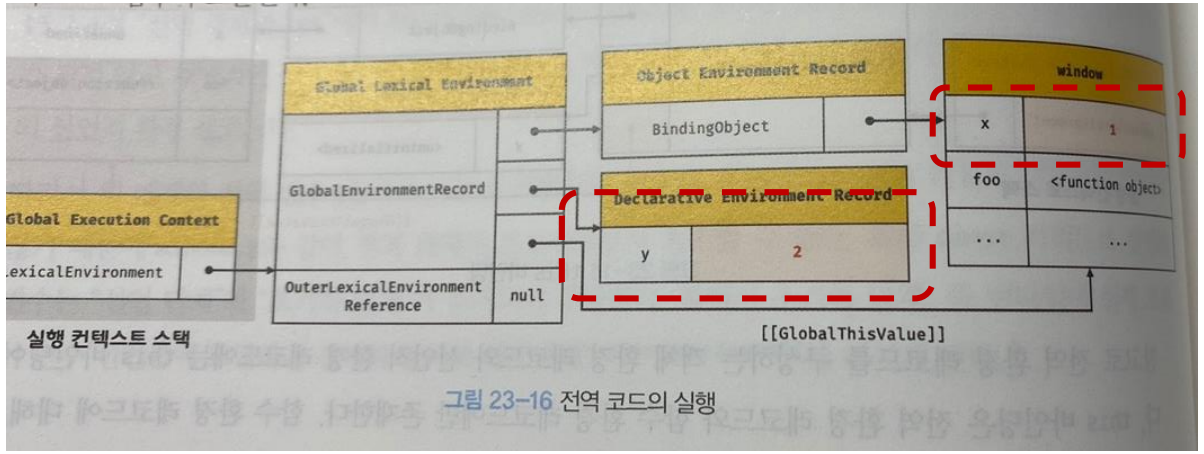


그림 23-16 전역 코드의 실행

# 실행 컨텍스트 생성 및 식별자 검색 과정

**4. foo 함수 코드 실행 및 평가**

4.1. 함수 실행 컨텍스트 생성

4.2. 함수 렉시컬 환경 생성

4.2.1. 함수 환경 레코드 생성

4.2.2. this 바인딩

4.2.3. 외부 렉시컬 환경 참조 결정

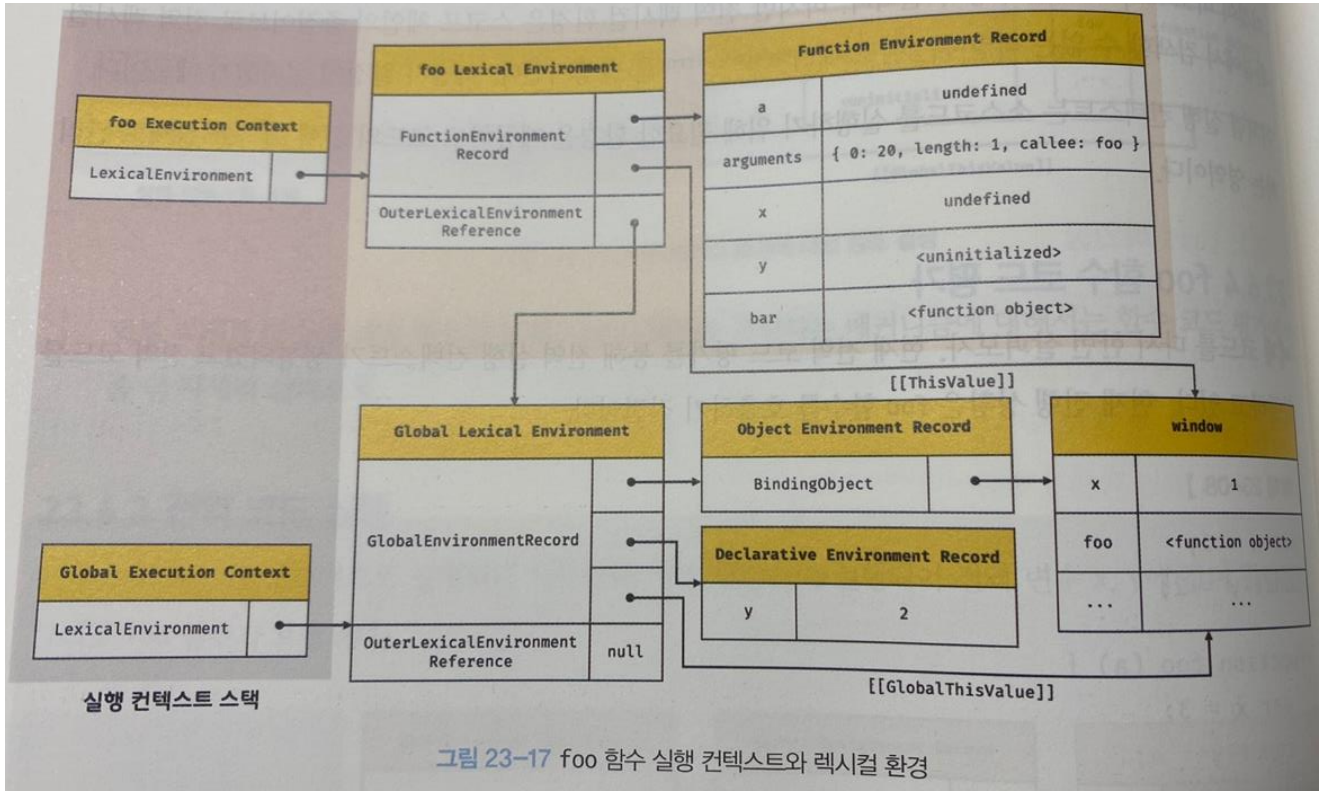



그림 23-17 foo 함수 실행 컨텍스트와 렉시컬 환경

```
1   var x = 1;
2   const y = 2;
3
4   function foo(a) {
5     var x = 3;
6     const y = 4;
7
8     function bar(b) {
9       const z = 5;
10      console.log(a + b + x + y + z);
11    }
12
13    bar(10);
14  }
15
16  foo(20); // 42
```

# 실행 컨텍스트 생성 및 식별자 검색 과정

## 4.1. 함수 실행 컨텍스트 생성

실행 컨텍스트 스택에 함수 실행 컨텍스트는 **함수 렉시컬 환경이 완성된 다음** 실행 컨텍스트 스택에 푸시
(전역 실행 컨텍스트는 실행 컨텍스트 스택에 비어있는 전역 실행 컨텍스트 생성 및 push)

## 4.2. 함수 렉시컬 환경 생성

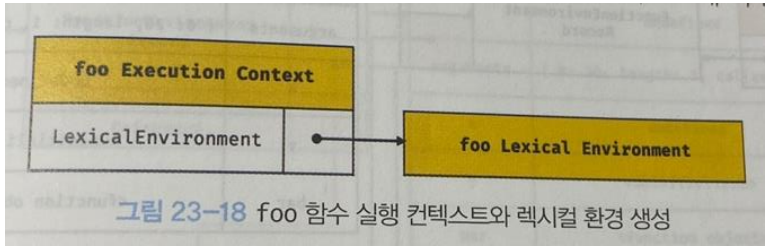


그림 23-18 foo 함수 실행 컨텍스트와 렉시컬 환경 생성

## 4.2.1 함수 환경 레코드 생성

매개변수, 지역 변수, aguments 객체, 중첩 함수 등록 및 관리

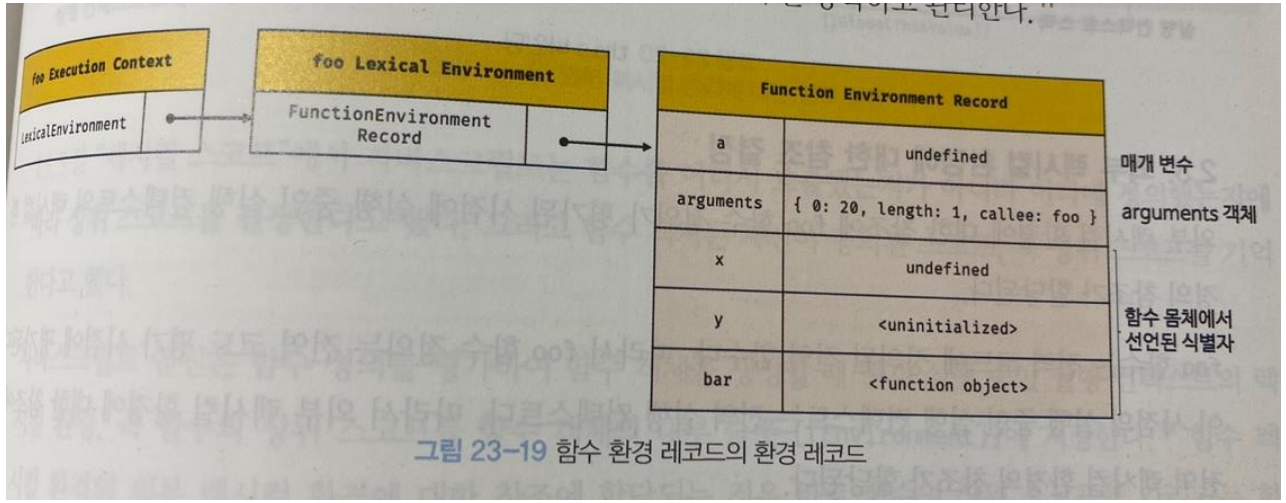


그림 23-19 함수 환경 레코드의 환경 레코드

# 실행 컨텍스트 생성 및 식별자 검색 과정

## 4.2.2 this바인딩

함수 환경 레코드의 [[ThisValue]] 내부 슬롯에 this가 바인딩
foo 함수는 일반 함수로 호출되었으므로, this는 전역 객체를 참조

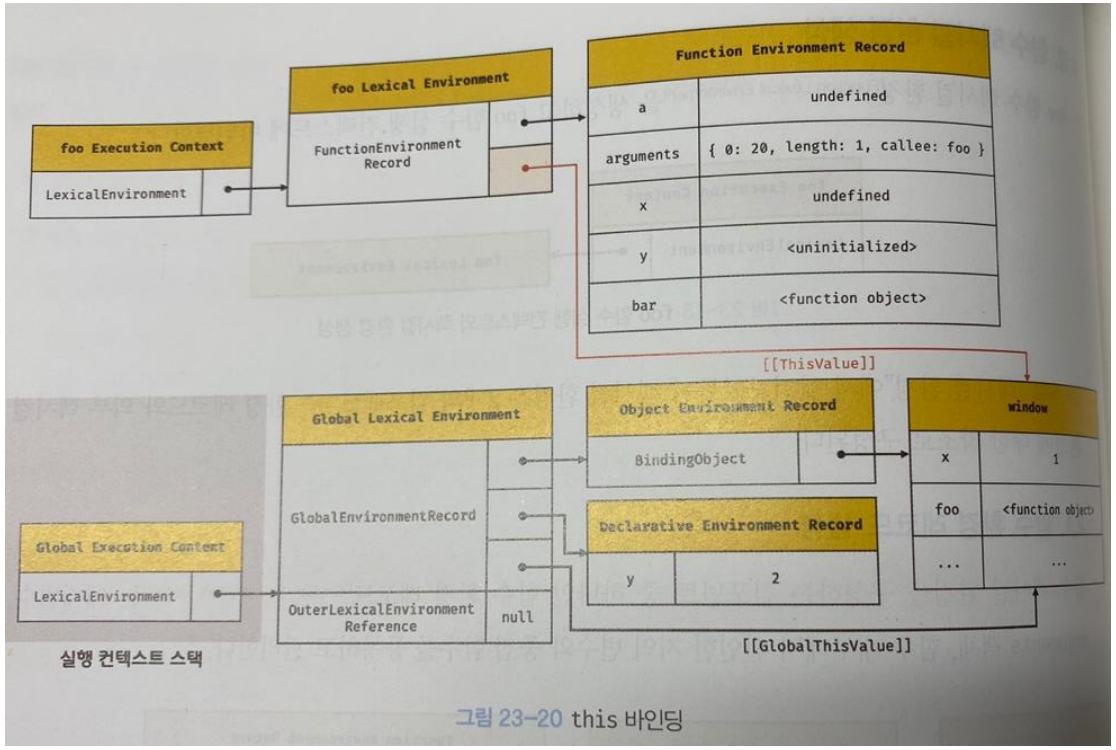



그림 23-20 this 바인딩

# 실행 컨텍스트 생성 및 식별자 검색 과정

**4.2.3 외부 렉시컬 환경에 대한 참조 결정**

foo 함수 정의가 평가된 시점에 실행 중인 실행 컨텍스트의 렉시컬 환경의 참조가 할당
foo 함수는 전역 코드에 정의된 전역 함수이며, foo 함수 정의는 전역 코드 평가 시점에 평가
그러므로, 외부 렉시컬 환경에 대한 참조에는 전역 렉시컬 환경 참조가 할당

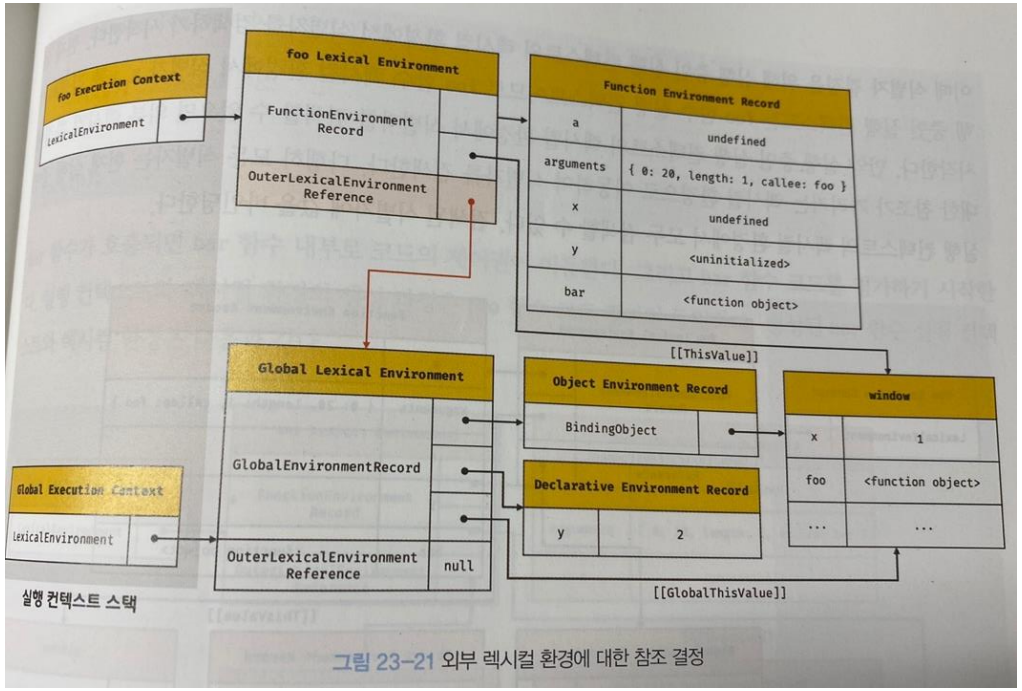


그림 23-21 외부 렉시컬 환경에 대한 참조 결정

함수 객체의 [[Environment]] 내부 슬롯에 함수의 상위 스코프 저장
함수 렉시컬 환경의 외부 렉시컬 환경에 대한 참조에 할당되는 것은 함수 객체의 [[Environment]] 내부 슬롯 값