

this, 실행 컨텍스트, 클로저

6 / this, 실행 컨텍스트, 클로저



목차

1. this란?
2. 실행 컨텍스트
3. 클로저
4. 여러 js 파일 import 해와서 참고하기
5. 동기, 비동기
6. 콜백지옥, Promise 맛보기

01

this 란?

📌 this

- this는 자신이 속한 객체 또는 자신이 생성할 인스턴스를 가리키는 자기 참조 변수
- this를 통해 자신이 속한 객체 또는 자신이 생성할 인스턴스의 프로퍼티나 메서드를 참조할 수 있음
- this가 가리키는 값은 함수 호출 방식에 의해 동적으로 결정된다

```
const circle = {  
  radius: 5,  
  getDiameter() {  
    // 이 메서드가 자신이 속한 객체의 프로퍼티나 다른 메서드를 참조하려면  
    // 자신이 속한 객체인 circle을 참조할 수 있어야 한다.  
    // 하지만 자기를 재귀적으로 참조하는 방식은 바람직하지 않음  
    return 2 * circle.radius;  
  }  
};  
  
console.log(circle.getDiameter()); // 10
```

this, 실행 컨텍스트, 클로저

Ⓢ this

- this는 어디서 호출하느냐에 따라 값이 달라진다

함수 호출 방식	this가 가리키는 값(this 바인딩)
일반 함수로서 호출	전역 객체
메서드로서 호출	메서드를 호출한 객체(마침표 앞의 객체)
생성자 함수로서 호출	생성자 함수가 (미래에) 생성할 인스턴스
apply/call/bind 메서드에 의한 간접 호출	apply/call/bind 메서드에 첫번째 인수로 전달한 객체

this, 실행 컨텍스트, 클로저

☑ this

화살표 함수 this

- 함수 자체의 this 바인딩을 갖지 않는다 (자체적으로 this가 가리키는 값이 없다)
 - 따라서 스코프 체인을 통해 상위 스코프의 this를 참조한다.
 - 화살표 함수 내부에서 this를 참조하면 상위 스코프의 this를 그대로 참조한다: lexical this

this, 실행 컨텍스트, 클로저

📌 this

- call, apply, bind
 - this를 원하는 객체로 변경할 수 있도록 해주는 함수
 - 각각 차이점
 - call
 - 모든 함수에서 사용가능하며, this를 특정 객체로 지정할 수 있다.
 - 호출함수.call(this로 지정할 대상, 추가 매개변수);
 - apply
 - call과 유사하지만 this 이후의 매개변수를 배열형태로 받는다
 - 호출함수.call(this로 지정할 대상, 추가 매개변수(배열형태));
 - bind
 - call, apply와 달리 새로운 함수를 만들어 반환해줌
 - let ??? = 호출함수.bind(this로 지정할 대상);

02

실행 컨텍스트

this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

- 실행 컨텍스트란?
 - 실행할 코드에 제공할 환경 정보들을 모아 놓은 객체
 - 소스코드는 4가지 타입이 있고, 실행 컨텍스트도 이에 따라 나뉜다.

소스코드의 타입	설명
전역 코드	전역에 존재하는 소스코드를 말한다. 전역에 정의된 함수, 클래스 등의 내부 코드는 포함되지 않는다
함수 코드	함수 내부에 존재하는 소스코드를 말한다. 함수 내부에 중첩된 함수, 클래스 등의 내부 코드는 포함되지 않는다.
eval 코드	빌트인 전역 함수인 eval 함수에 인수로 전달되어 실행되는 소스코드를 말한다.
모듈 코드	모듈 내부에 존재하는 소스코드를 말한다. 모듈 내부의 함수, 클래스 등의 내부 코드는 포함되지 않는다.

this, 실행 컨텍스트, 클로저

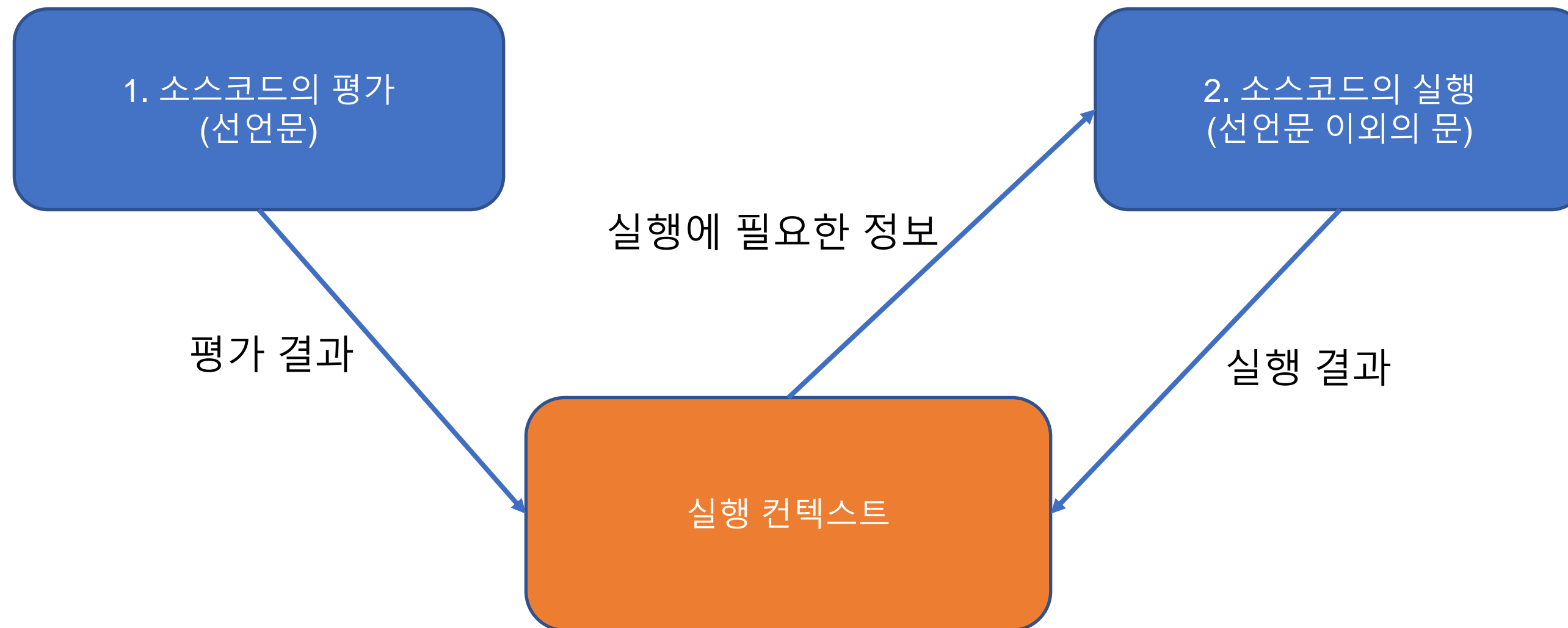
④ 실행 컨텍스트

<https://yongdanielliang.github.io/animation/web/Stack.html>

this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

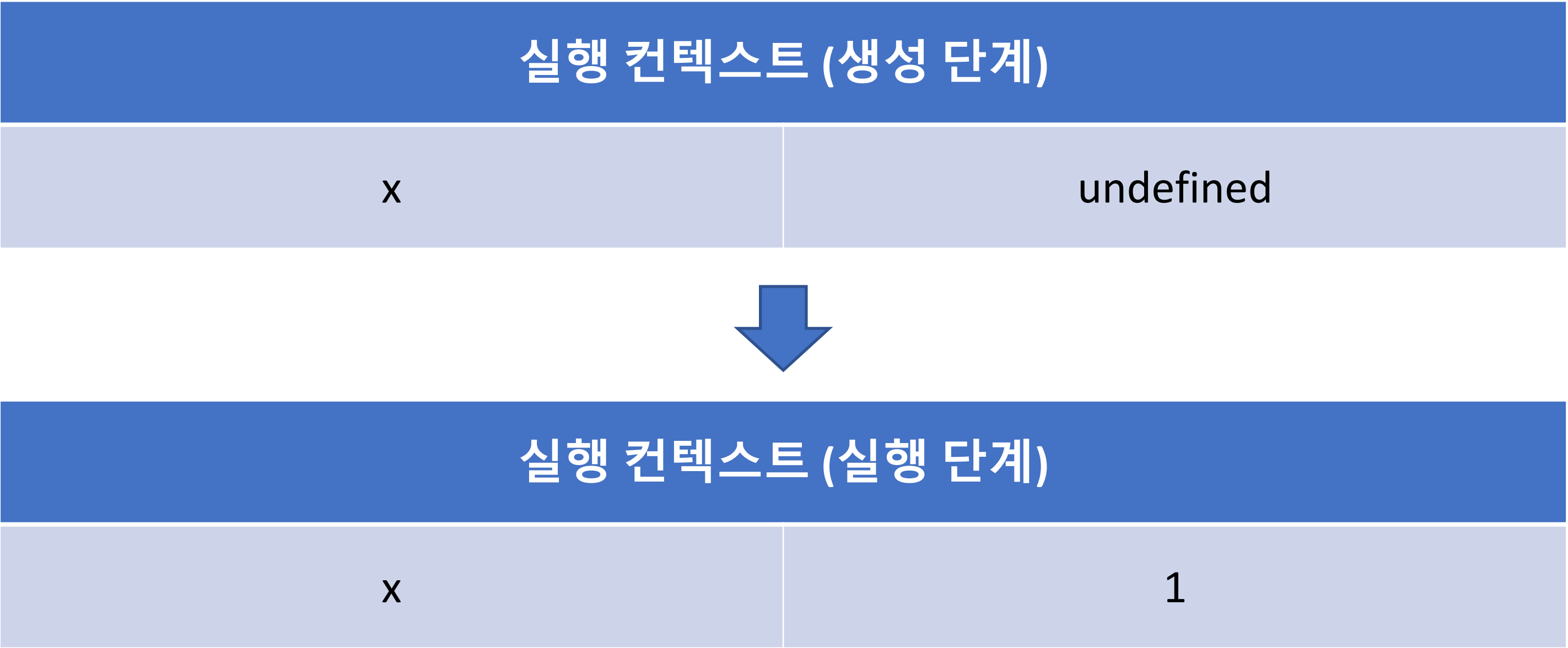
- 소스코드 평가와 실행



this, 실행 컨텍스트, 클로저

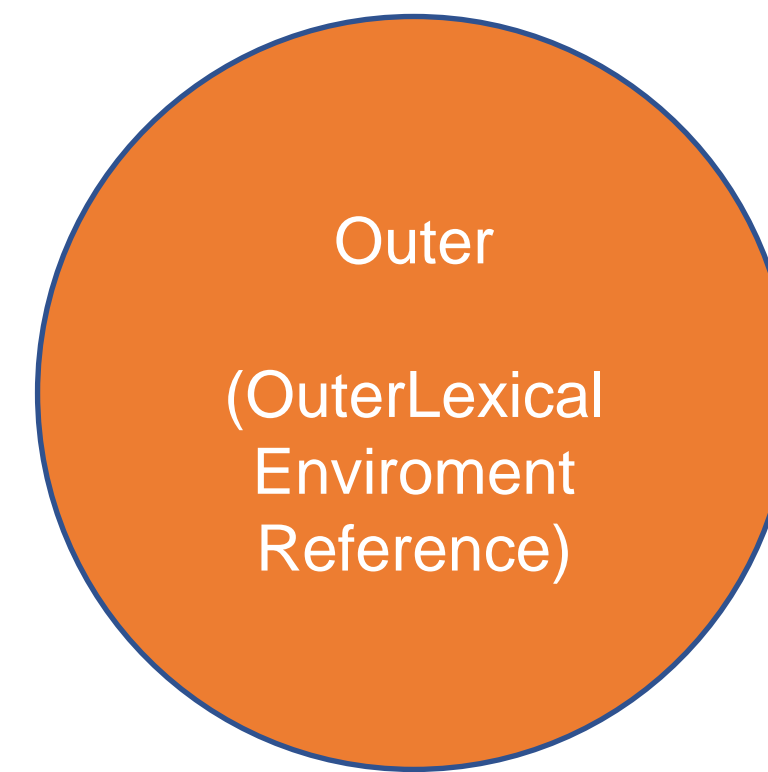
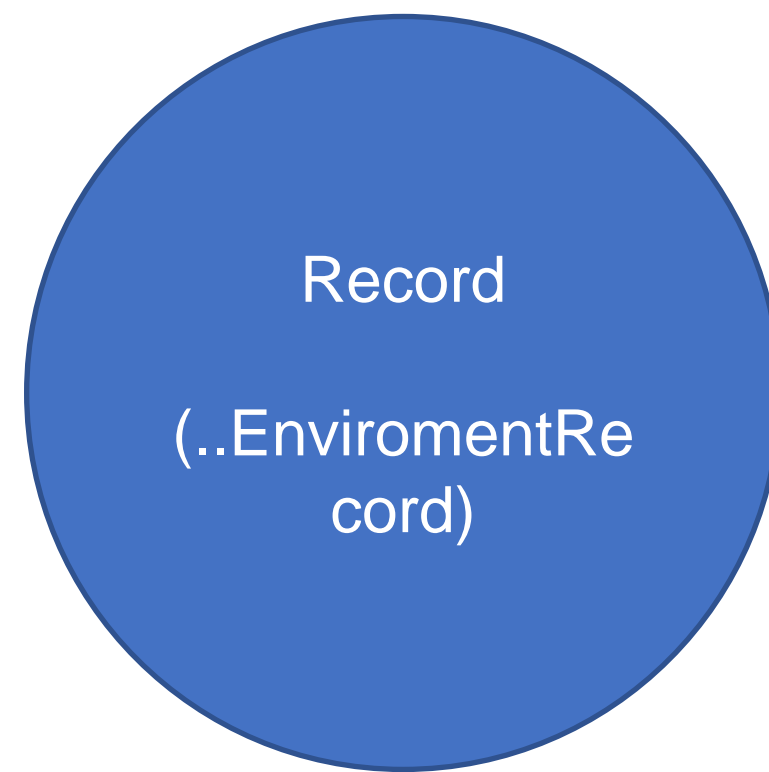
④ 실행 컨텍스트

- 소스코드 평가와 실행




this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

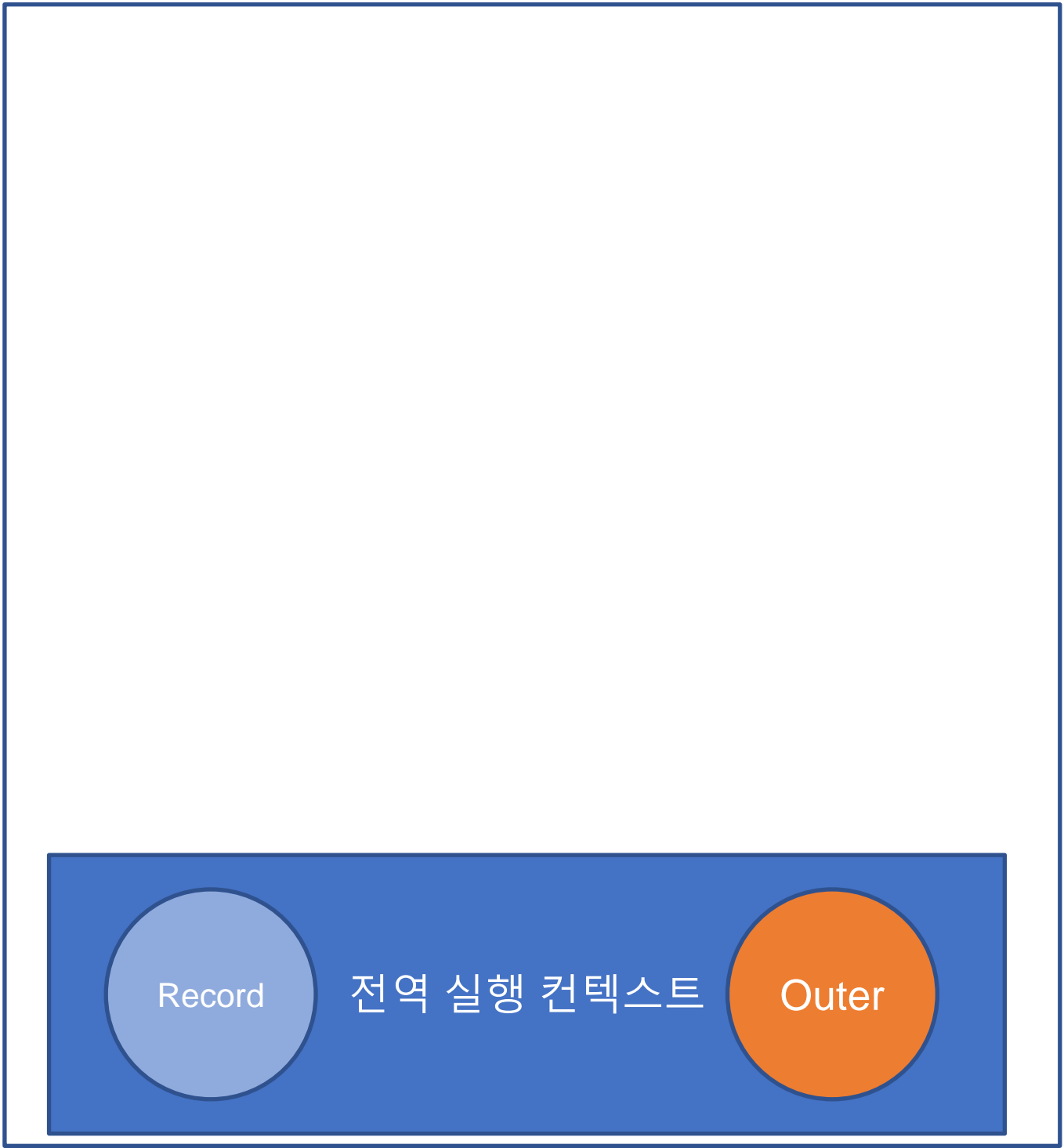


this, 실행 컨텍스트, 클로저

 JS 엔진: 전역을 지켜보겠어..


④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: A를 지켜보겠어..


④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: B를 지켜보겠어..


④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: A를 지켜보겠어..


④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



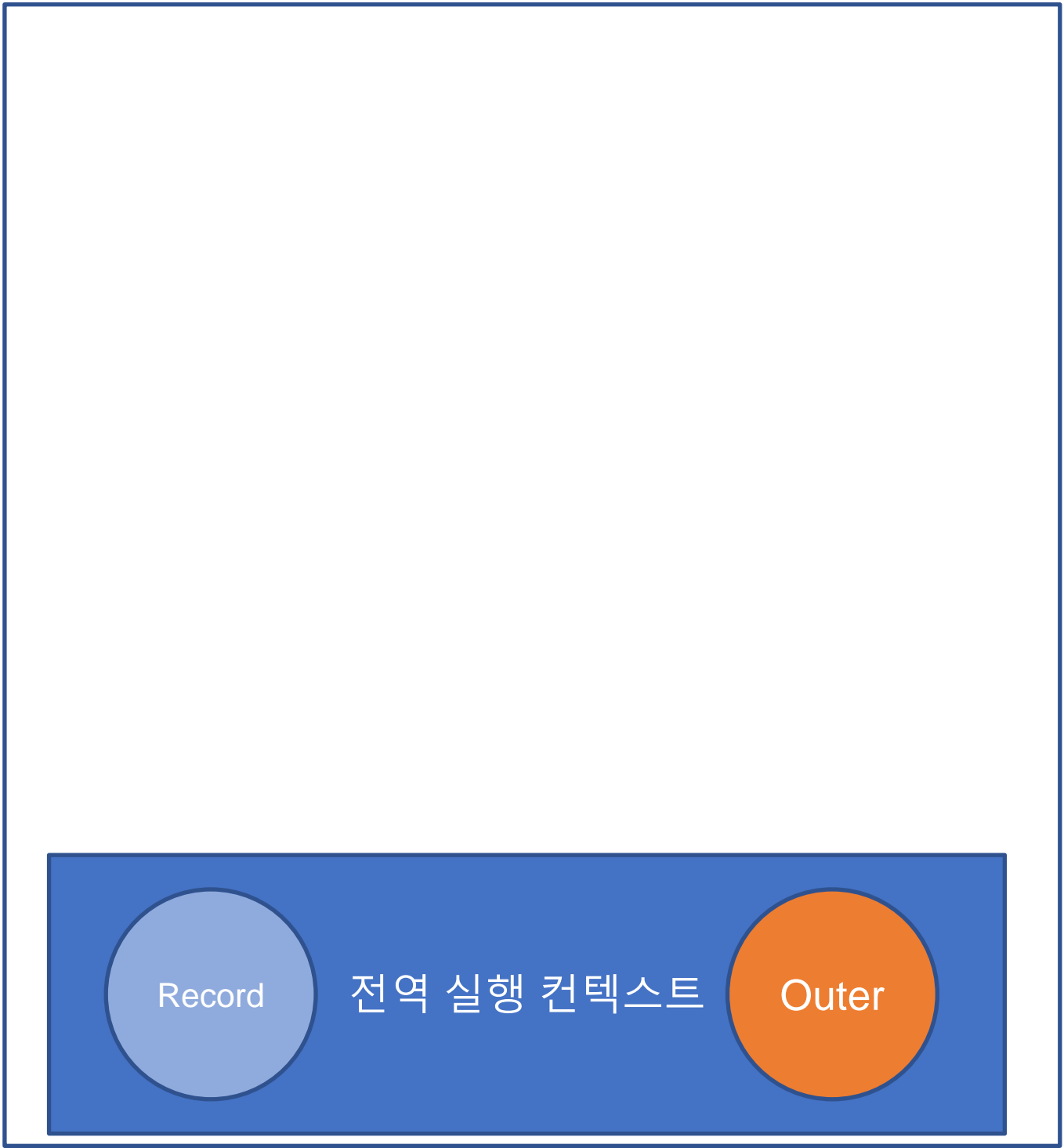
Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: 전역을 지켜보겠어..

④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: 더 이상 볼게 없군

④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



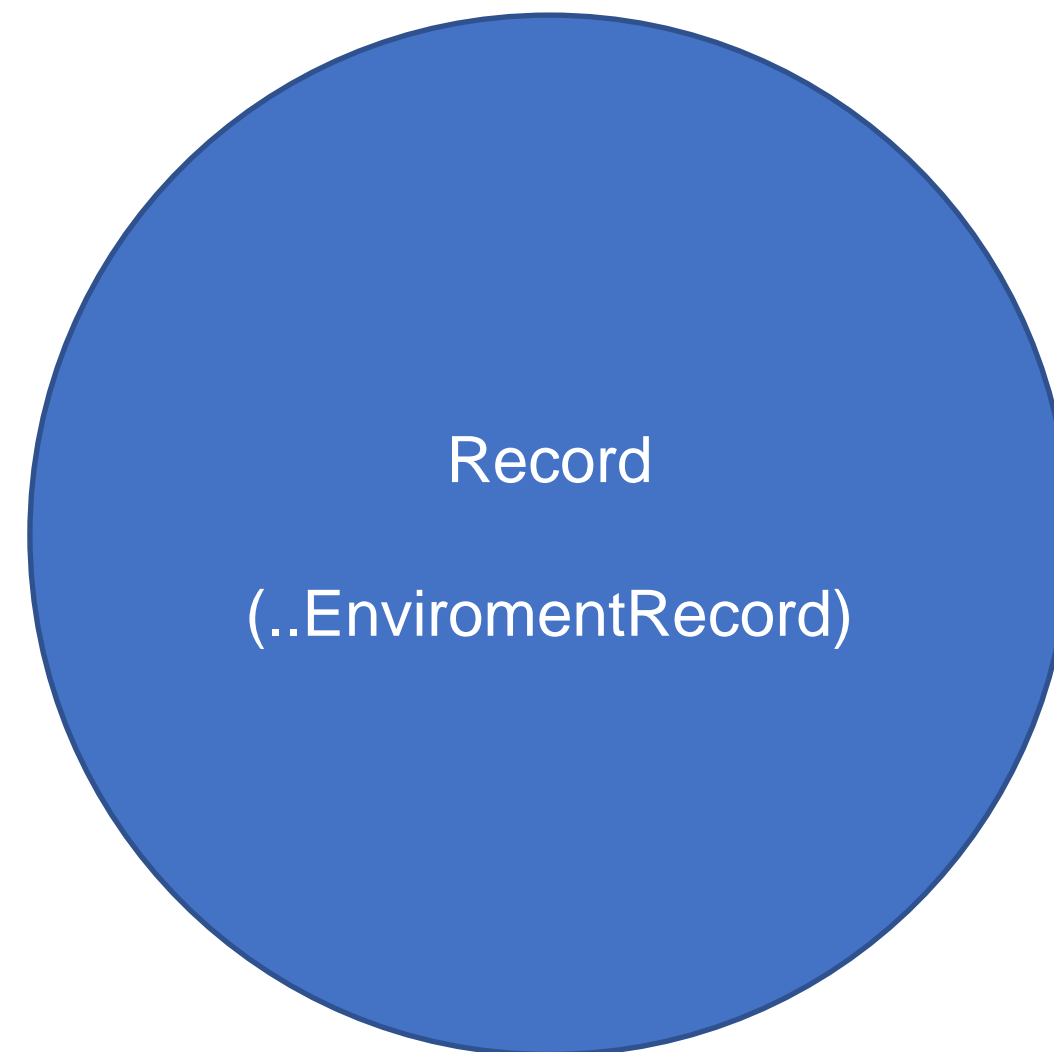
Call stack

this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

EnviromentRecord


➔ 식별자와 식별자에 바인딩 된 값을 기록해주는 객체



호이스팅 종류

- 변수 호이스팅
 - var
 - let, const
- 함수 호이스팅

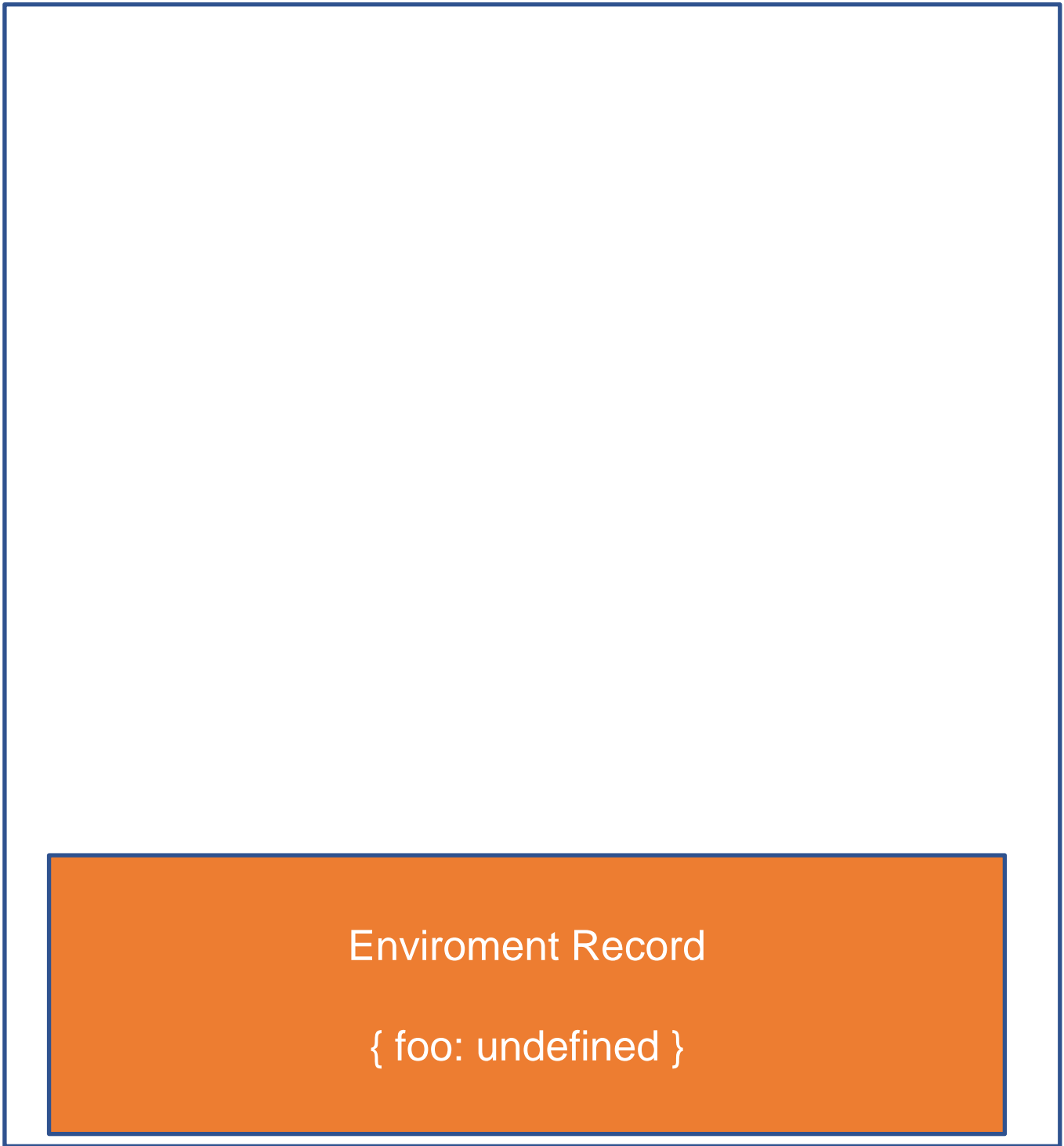
this, 실행 컨텍스트, 클로저

 JS 엔진: foo를 기록해라!

④ 실행 컨텍스트



```
console.log(foo);  
var foo = 'Hello';  
console.log(foo);
```




Call stack

this, 실행 컨텍스트, 클로저

④ 실행 컨텍스트

```
console.log(foo);  
var foo = 'Hello';  
console.log(foo);
```


 JS 엔진: 참고로 지금은 생성단계야

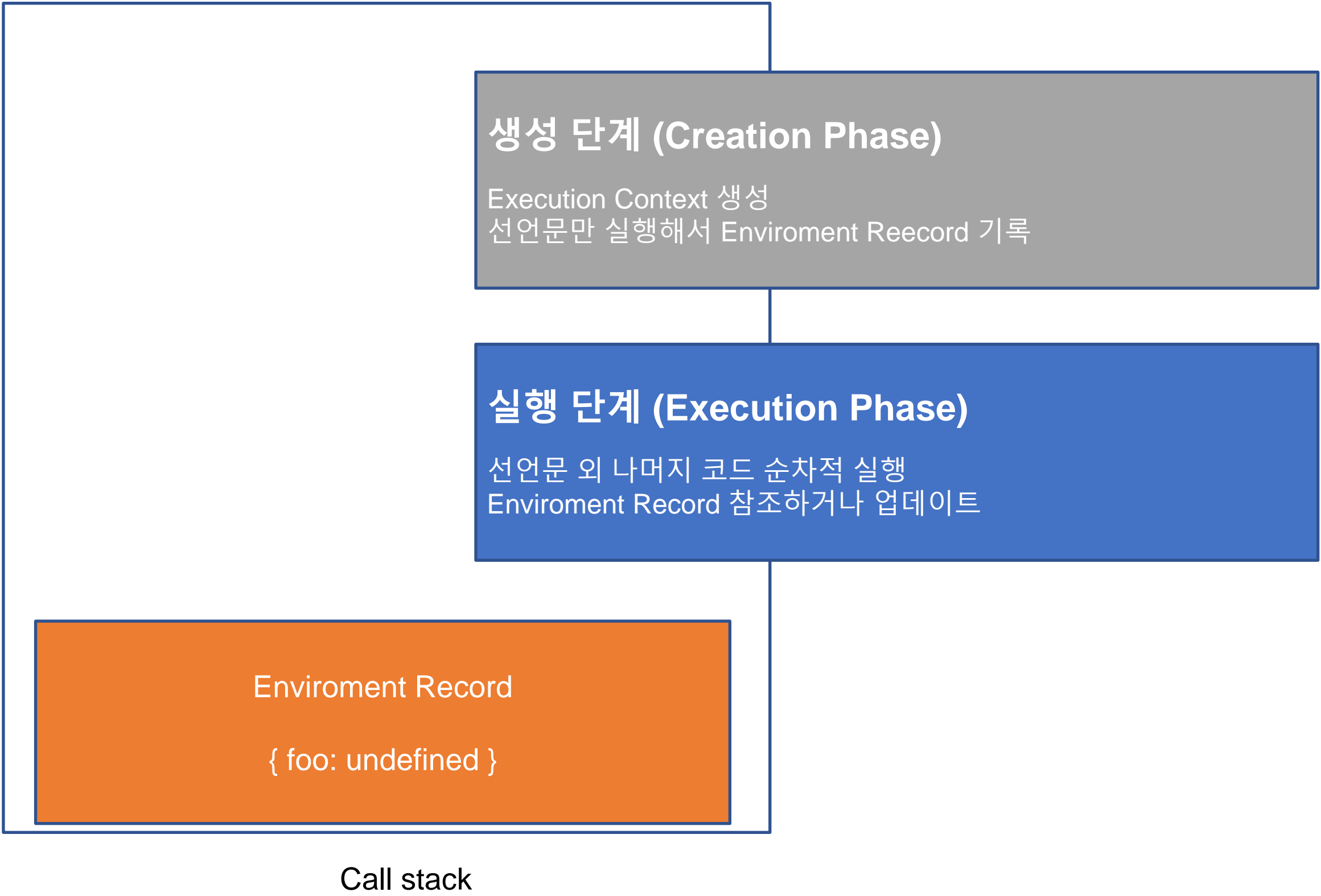


this, 실행 컨텍스트, 클로저

👉 실행 컨텍스트

```
console.log(foo);  
  
var foo = 'Hello';  
  
console.log(foo);
```

 JS 엔진: 지금은 실행단계야



this, 실행 컨텍스트, 클로저

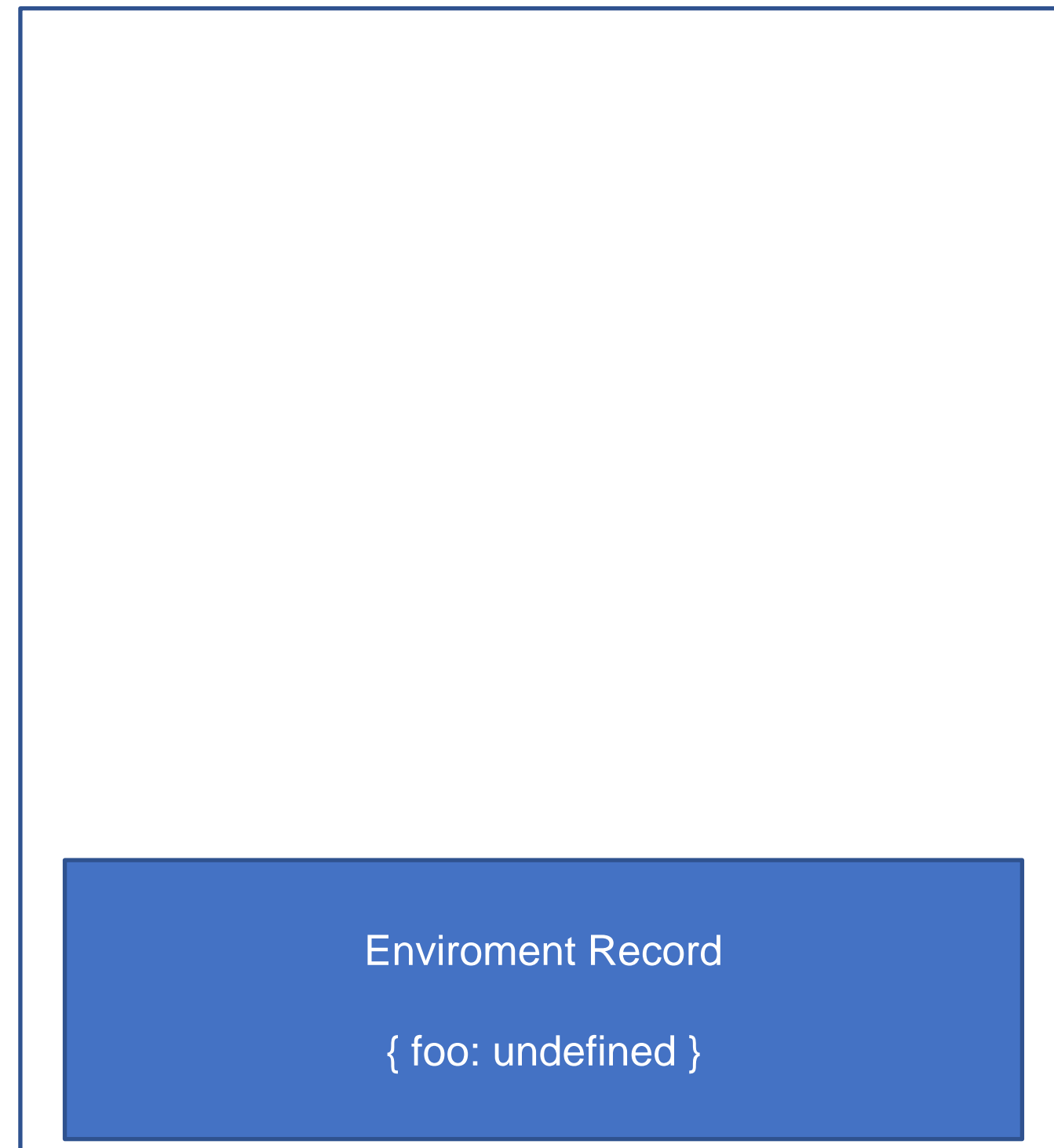


JS 엔진: foo가 뭐였더라? undefined 구만!

④ 실행 컨텍스트



```
console.log(foo);  
var foo = 'Hello';  
console.log(foo);
```



Call stack

this, 실행 컨텍스트, 클로저

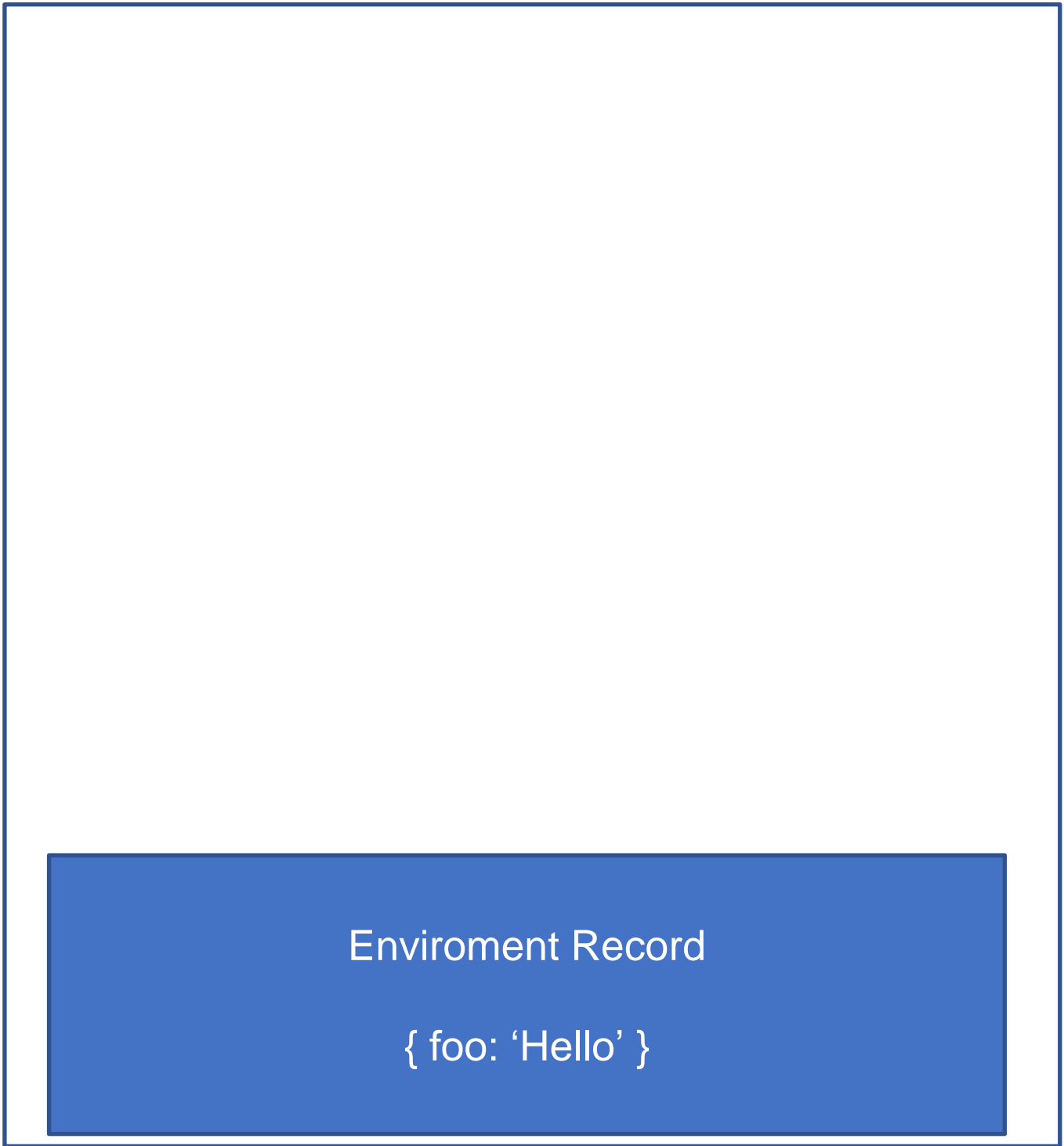


JS 엔진: foo를 변경시켜줘야 겠군

④ 실행 컨텍스트



```
console.log(foo);  
var foo = 'Hello';  
console.log(foo);
```



Call stack

this, 실행 컨텍스트, 클로저

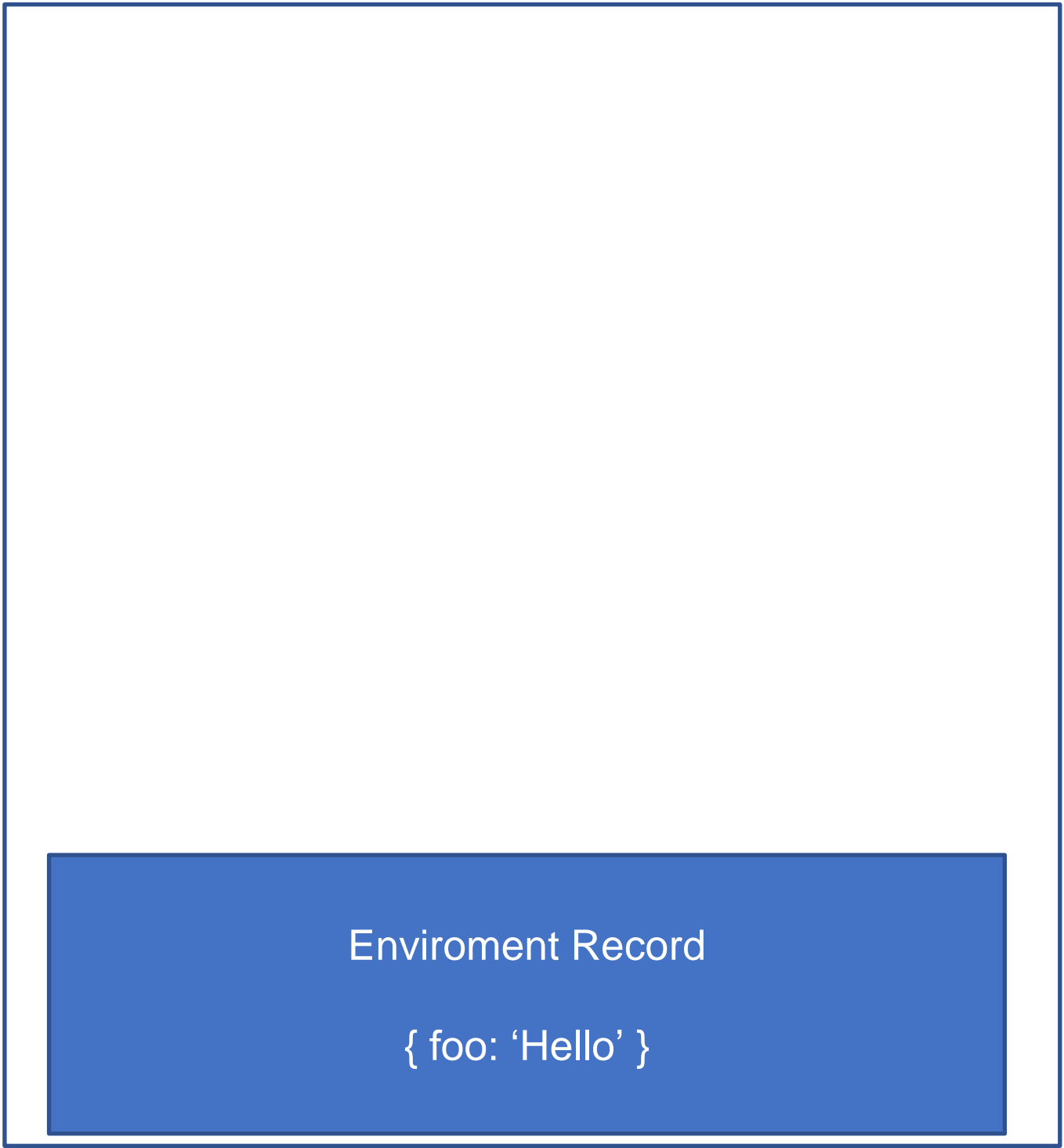
④ 실행 컨텍스트



```
console.log(foo);  
var foo = 'Hello';  
console.log(foo);
```



JS 엔진: foo가 뭐였더라? 'Hello' 구만!



Call stack

this, 실행 컨텍스트, 클로저

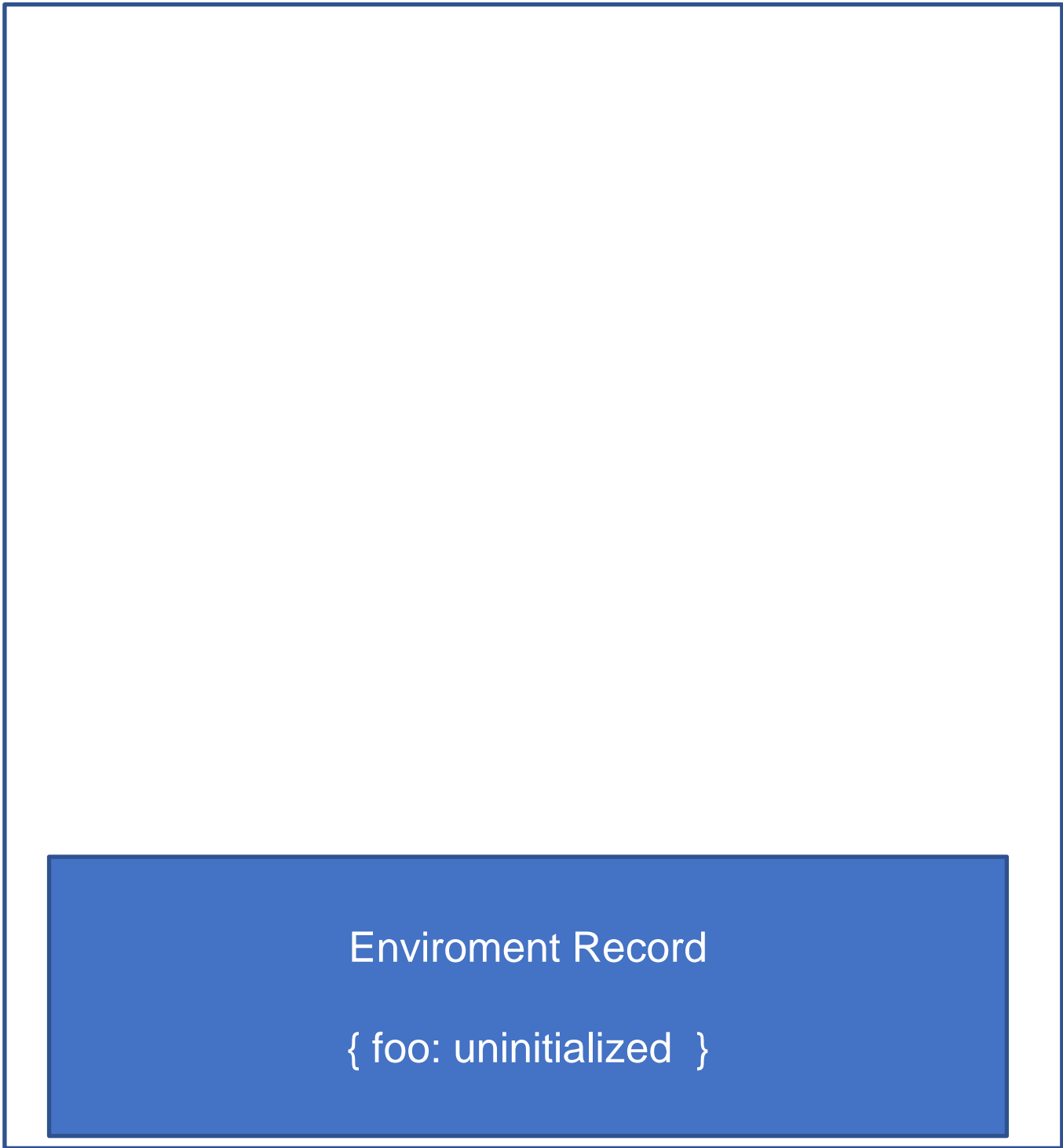
④ 실행 컨텍스트



```
console.log(foo);  
const foo = 'Hello';  
console.log(foo);
```



JS 엔진: foo가 뭐지?
일단 기록 해두자, 대신 값은 없어



Call stack

 JS 엔진: 예끼 이 양반아 foo는 쓸수 없어!

④ 실행 컨텍스트



this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

- 자바스크립트 엔진은 변수 선언을 다음과 같은 2단계에 거쳐 수행한다.
 - 생성단계
 - 선언 단계
 - 초기화 단계
 - 실행단계

선언 단계

초기화 단계

foo === undefined

선언 Declaration

메모리 공간을 확보한 다음 식별자와 연결

초기화 Initialization

식별자에 암묵적으로 undefined 값을 바인딩

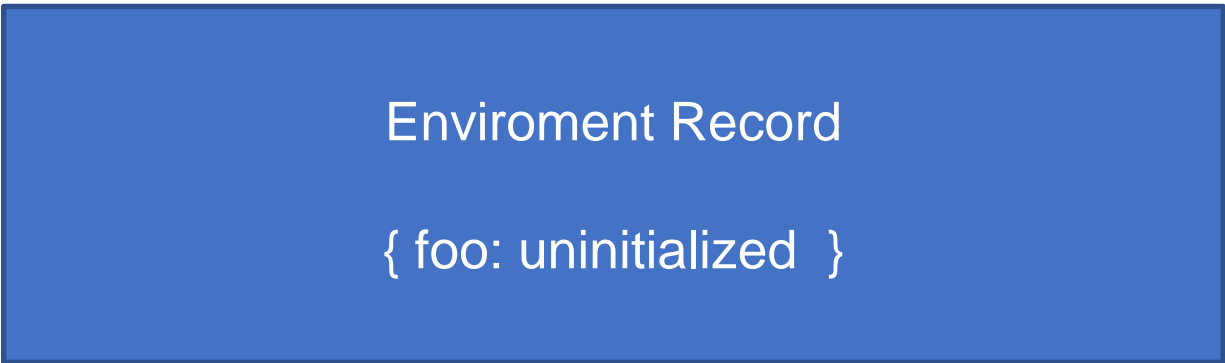
할당 단계

foo === 'Hello'

var 키워드로 선언한 변수 생명주기

④ 실행 컨텍스트

- 자바스크립트 엔진은 변수 선언을 다음과 같은 2단계에 거쳐 수행한다.
 - 생성단계
 - 선언 단계
 - 초기화 단계
 - 실행단계



선언 Declaration

메모리 공간을 확보한 다음 식별자와 연결

초기화 Initialization

식별자에 암묵적으로 undefined 값을 바인딩

let 키워드로 선언한 변수 생명주기

this, 실행 컨텍스트, 클로저

④ 실행 컨텍스트

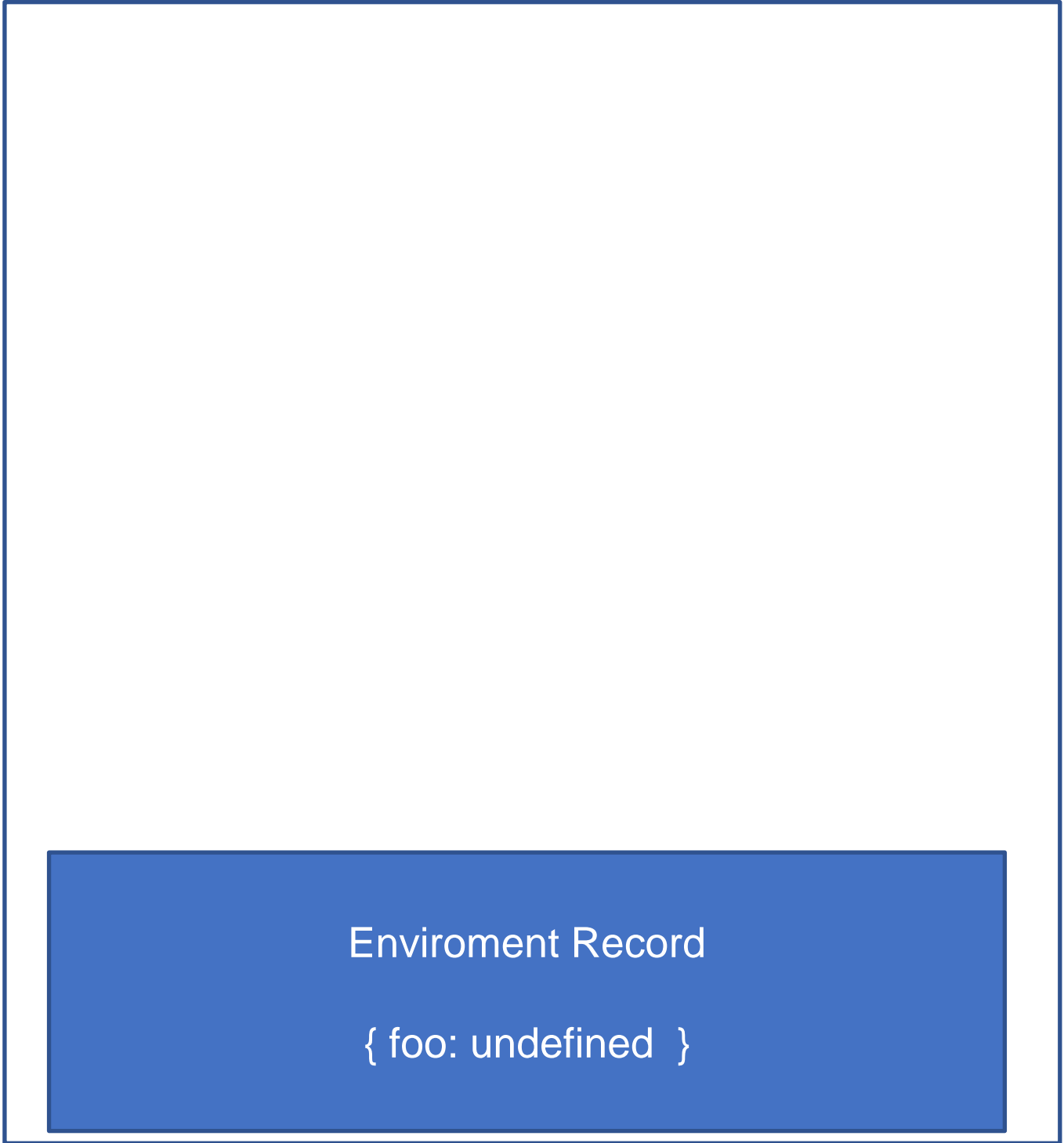
→

```
foo();

var foo = function() {
  // Hello
}
```



JS 엔진: 예끼 이 양반아 foo는 undefined인데 호출이 되나!
Type Error!!



Call stack

this, 실행 컨텍스트, 클로저



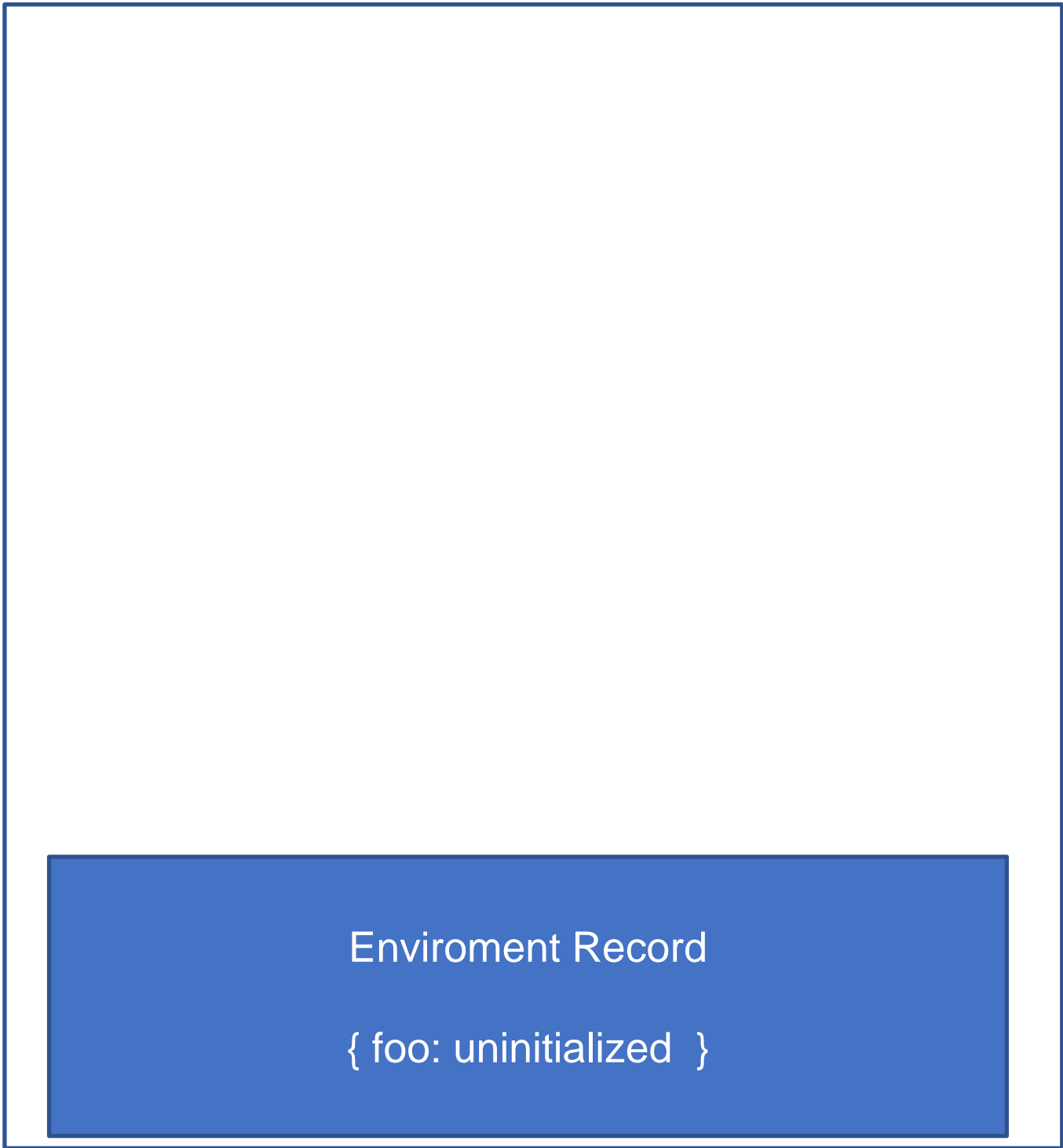
JS 엔진: 예끼 이 양반아 foo는 쓸수 없어! **Reference Error!!**

④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



```
foo( );  
  
let foo = function() {  
  // Hello  
}
```



Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: foo를 기록하자!

④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



```
foo();  
  
function foo() {  
  // do something..  
}
```

{ foo: f {} }

Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: foo를 실행시켜주자!

④ 실행 컨텍스트

실행 컨텍스트 생성 / 소멸 과정



```
foo();  
  
function foo() {  
  // do something..  
}
```

{ foo: f {} }

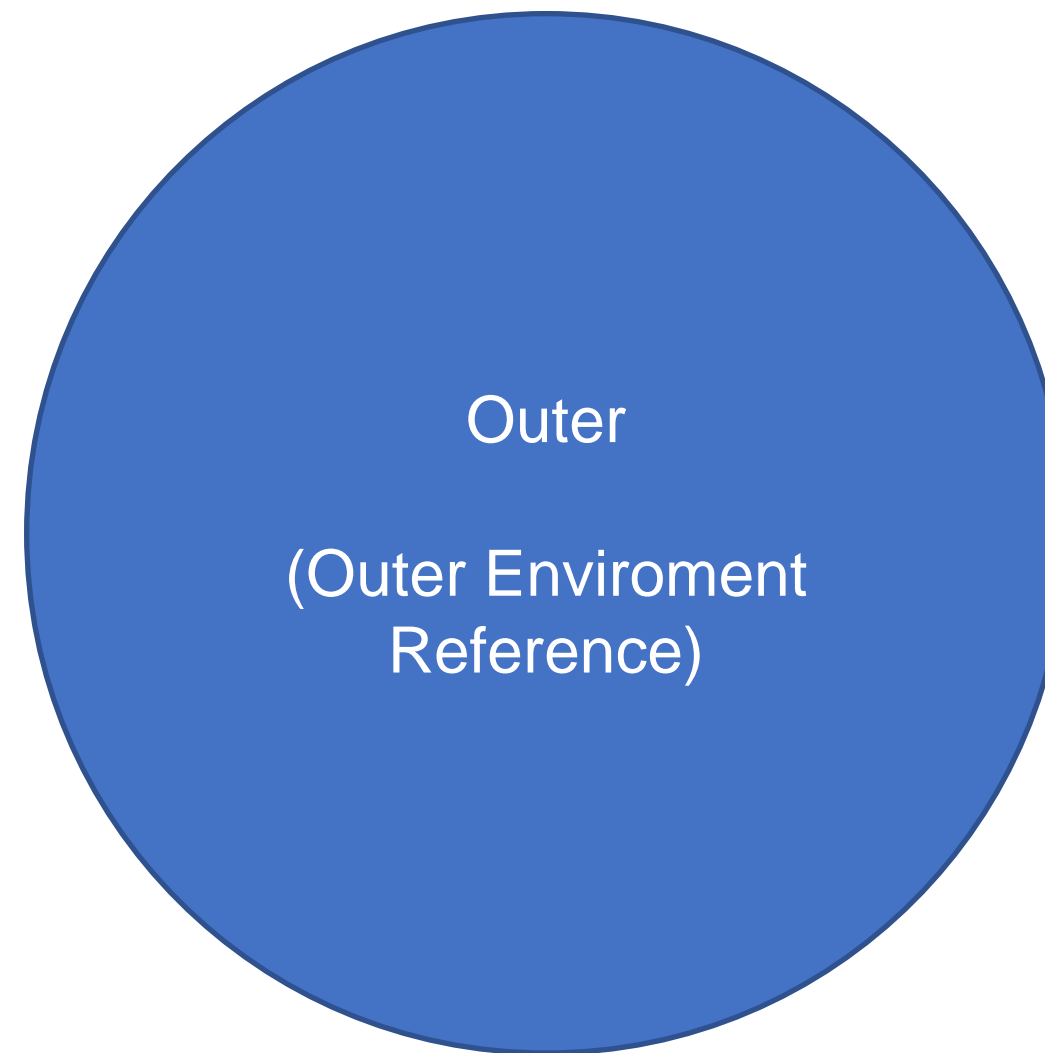
Call stack

this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

외부 환경 참조

바깥 Lexical Enviroment를 가리킴

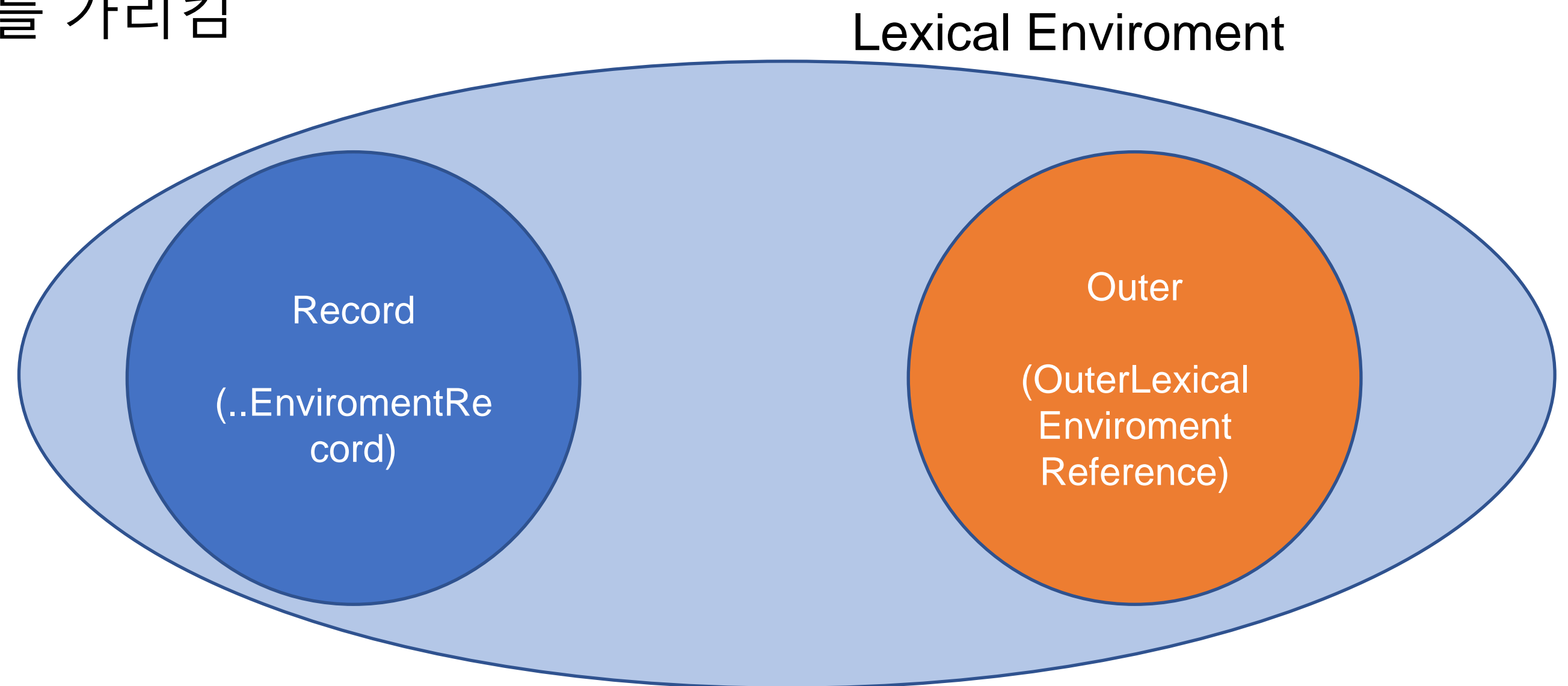


this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트

외부 환경 참조

바깥 Lexical Enviroment를 가리킴



this, 실행 컨텍스트, 클로저

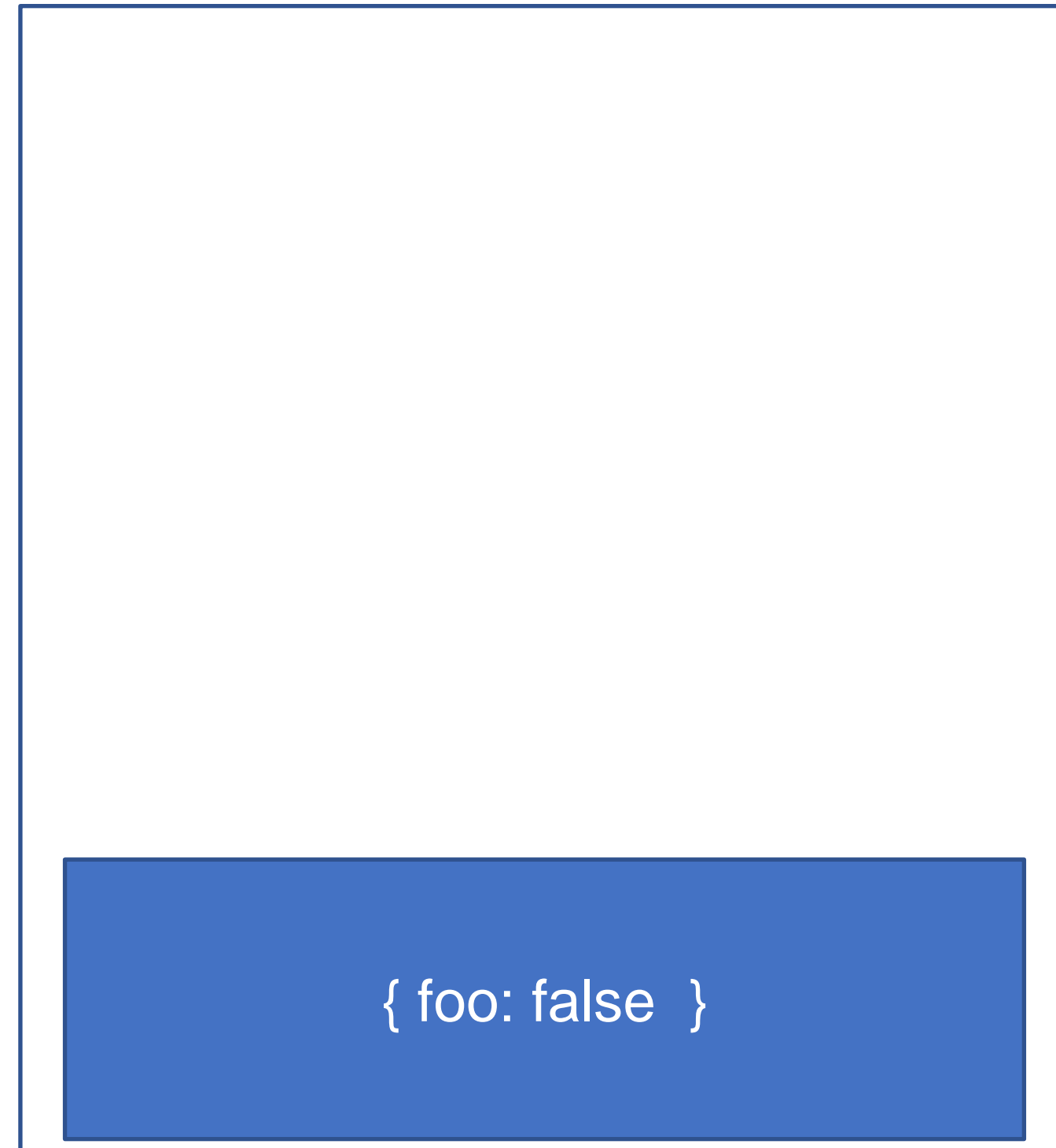
④ 실행 컨텍스트



```
let foo = false;  
console.log(foo);
```



JS 엔진: foo는 뭐였더라? false 지!



Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: myFunction 실행시켜!

☑ 실행 컨텍스트

```
let foo = false;  
  
function myFunction() {  
  let foo = true;  
  
  console.log(foo);  
}
```



```
myFunction();
```

{ foo: false, myFunction: f {} }

Call stack



JS 엔진: Outer 통로를 만들어주지!

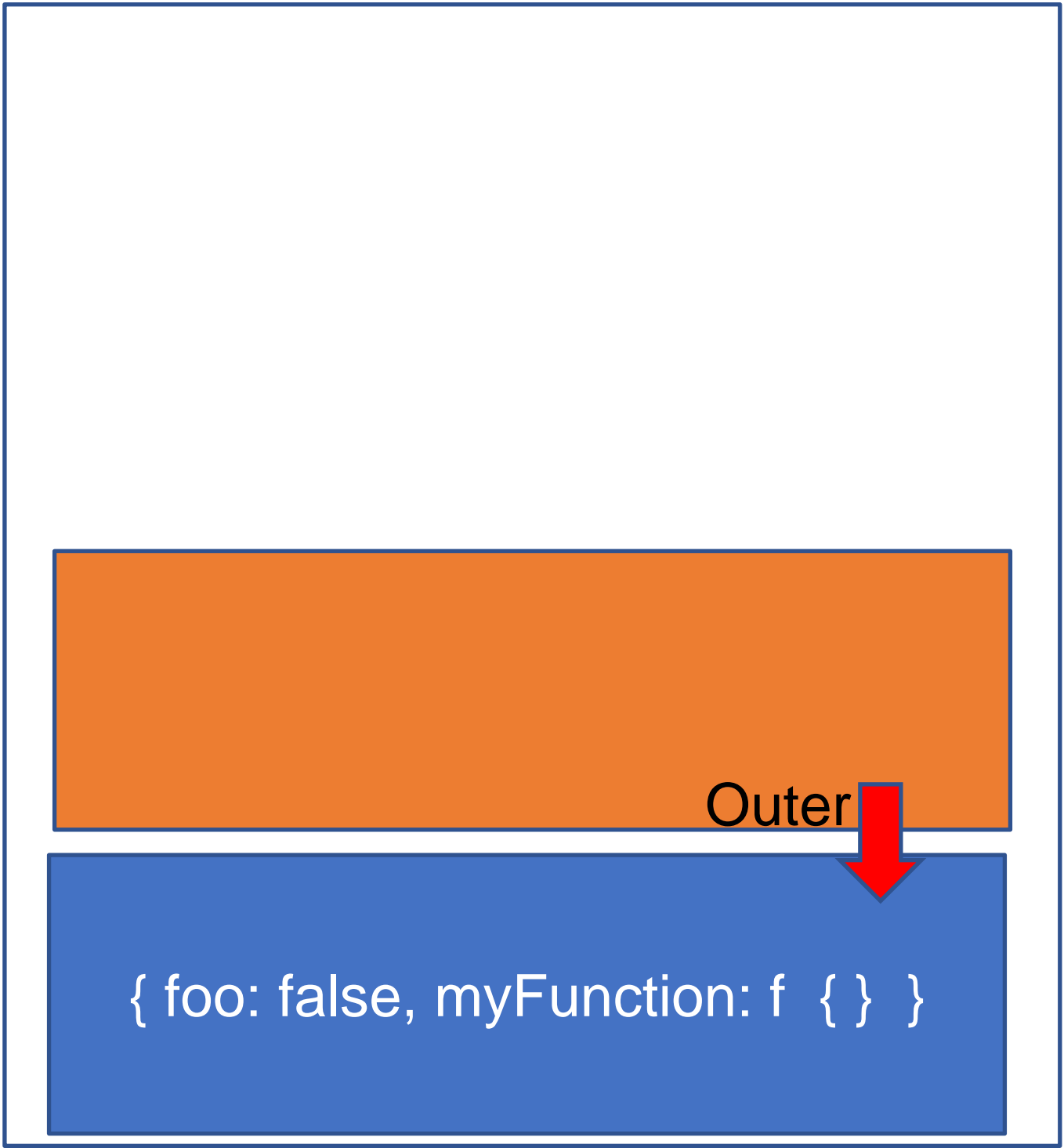
④ 실행 컨텍스트

```
let foo = false;

function myFunction() {
  let foo = true;

  console.log(foo);
}

myFunction();
```



Call stack

 JS 엔진: 무슨 값을 써야 되지..??

④ 실행 컨텍스트

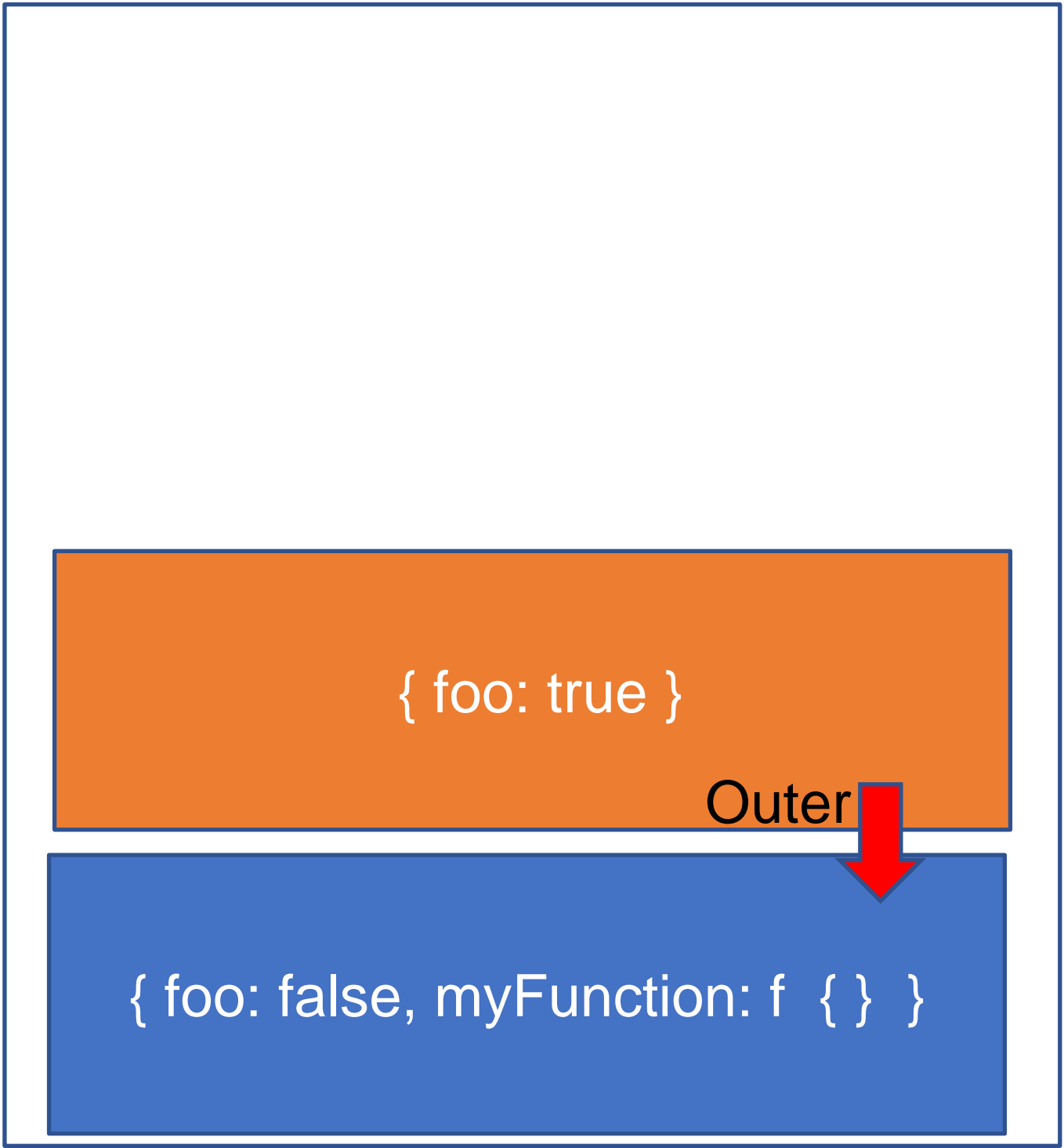


```
let foo = false;

function myFunction() {
  let foo = true;

  console.log(foo);
}

myFunction();
```



식별자 결정
코드에서 변수나 함수의 값을 결정하는 것

Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: Outer 통로를 만들어주지!!

☑ 실행 컨텍스트

```
let foo = false;

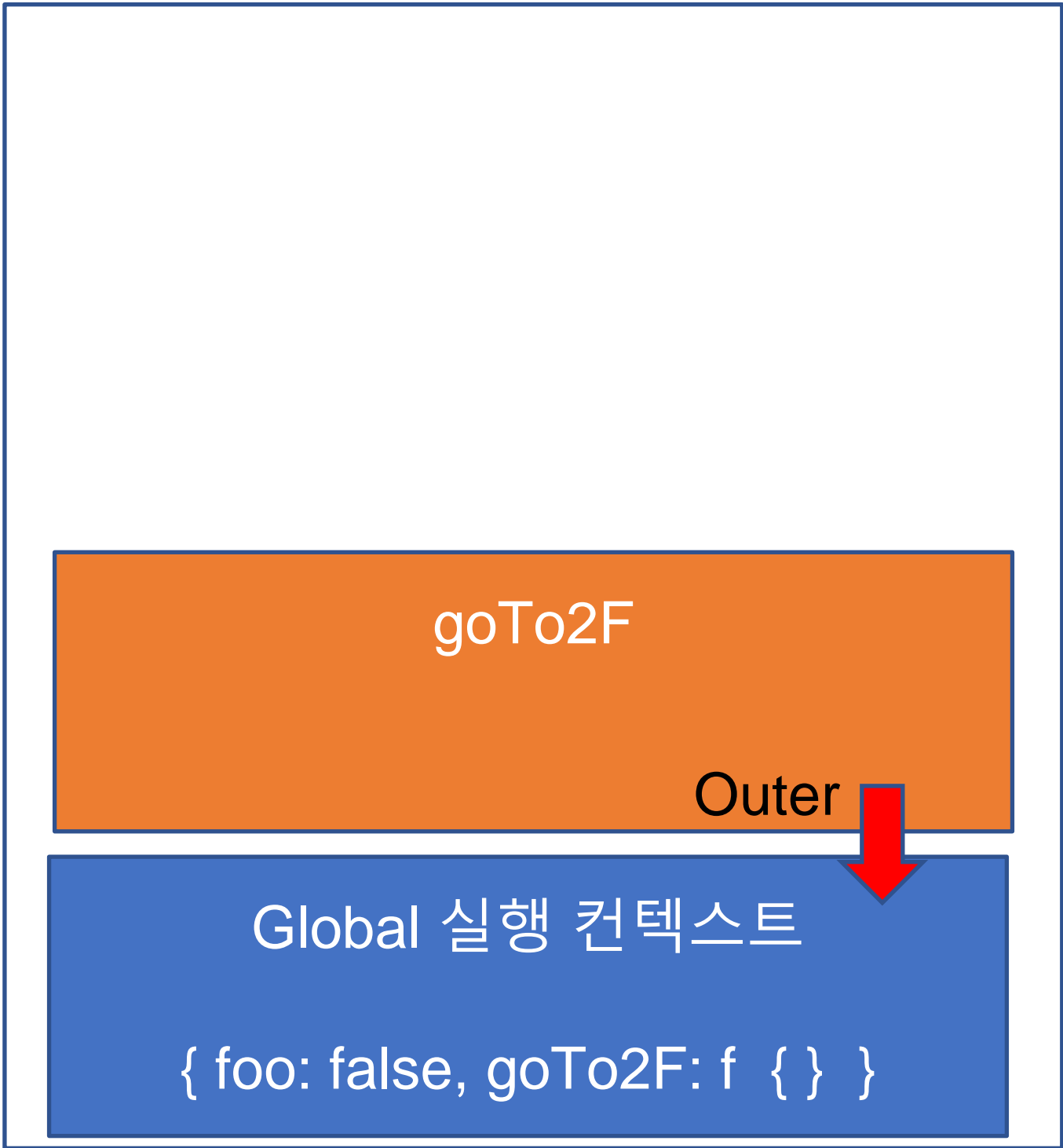
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
  }

  goTo3F();
}

goTo2F();
```



Call stack



JS 엔진: Outer 통로를 만들어주지!!

☑ 실행 컨텍스트

```
let foo = false;

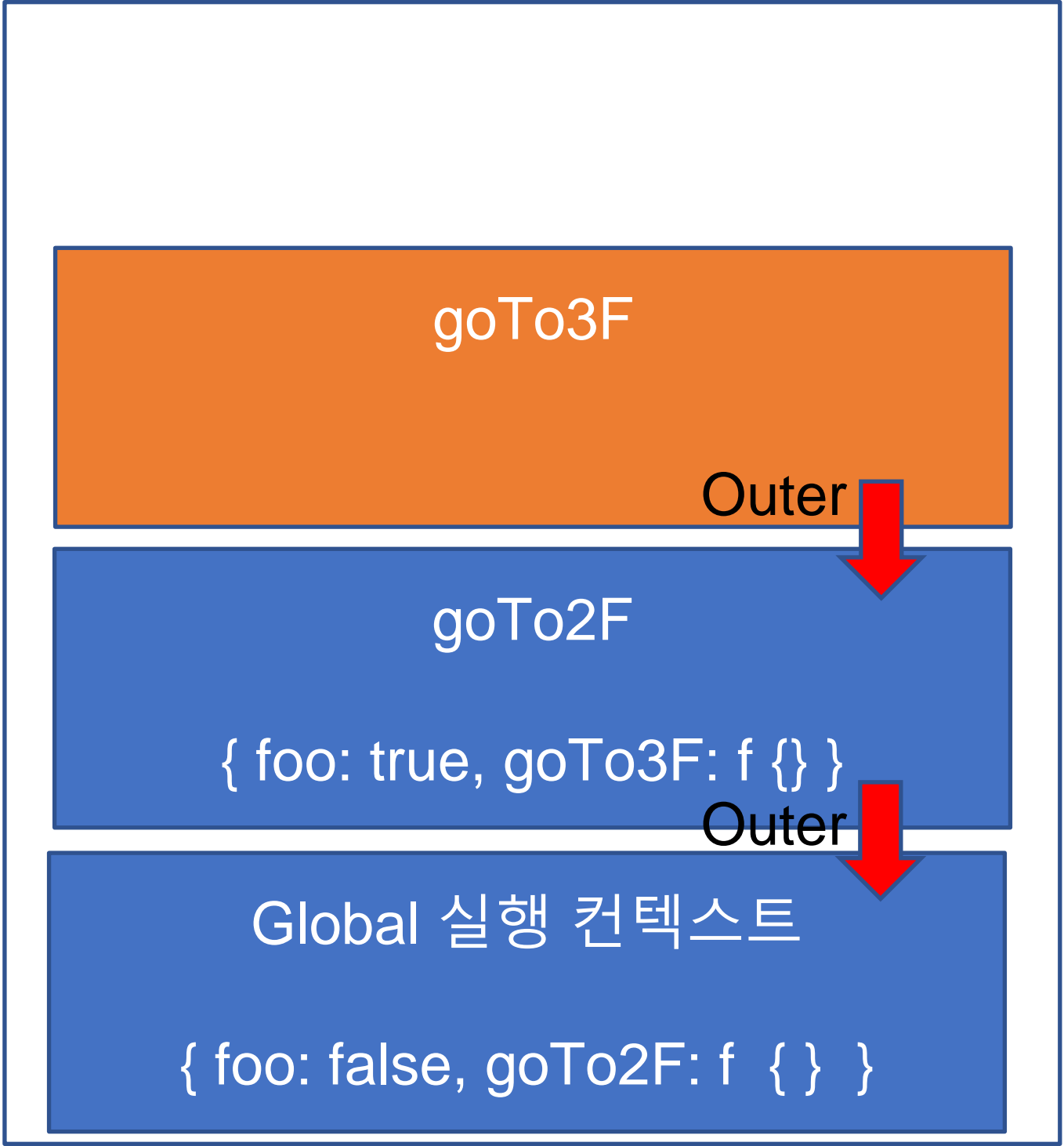
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
  }


  goTo3F();
}

goTo2F();
```



Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: bar? 현재 활성화된 거에서 찾아야지!

☑ 실행 컨텍스트



```
let foo = false;

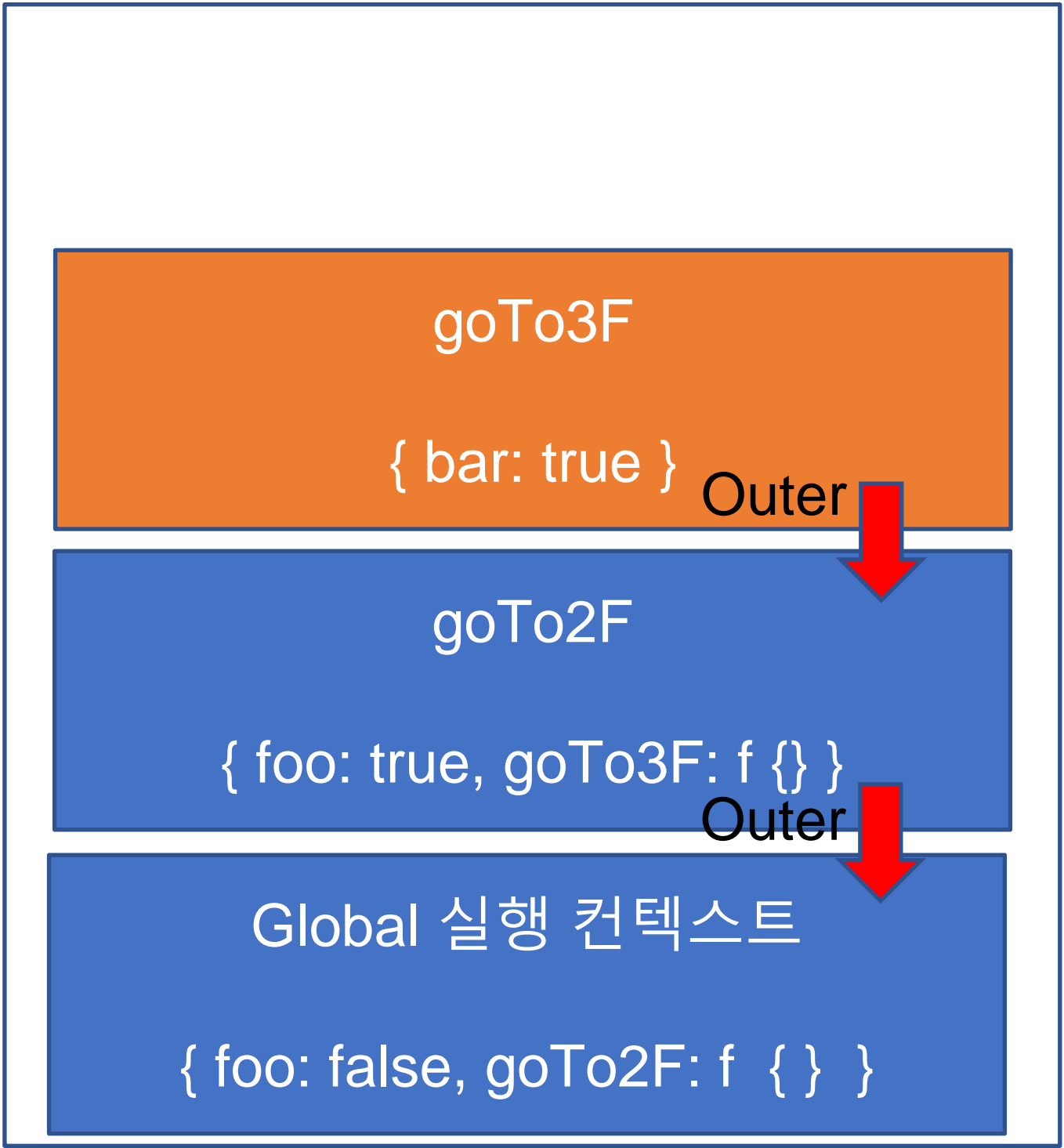
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;


    console.log(bar);
  }

  goTo3F();
}

goTo2F();
```



Call stack

 JS 엔진: fooBar? 어디에 뒀지?

☑ 실행 컨텍스트



```
let foo = false;

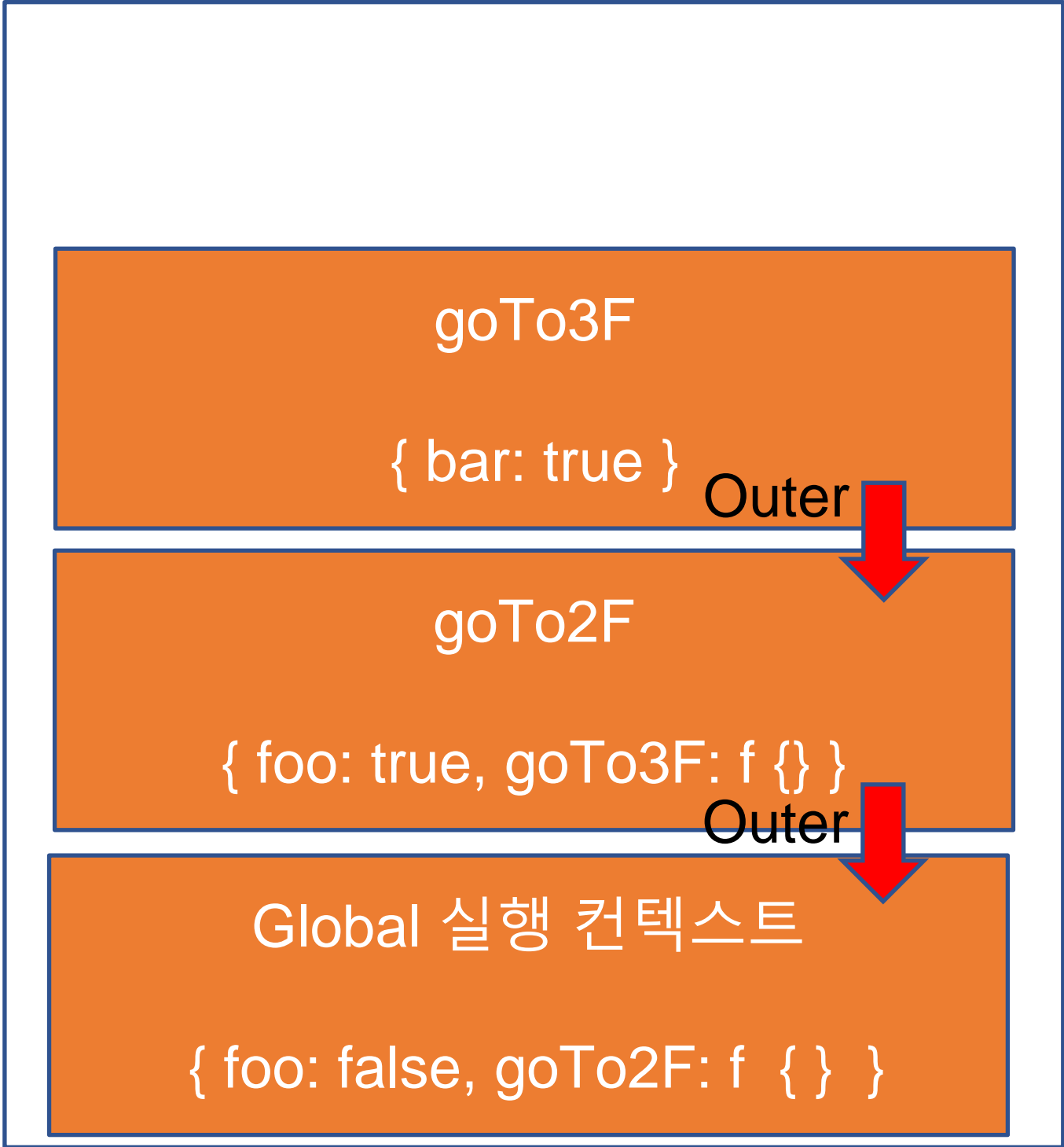
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
    console.log(fooBar);
  }

  goTo3F();
}

goTo2F();
```



Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: foo? 현재 활성화된 거에서 찾아야지!
근데 현재 실행 컨텍스트에는 없네..

☑ 실행 컨텍스트



```
let foo = false;

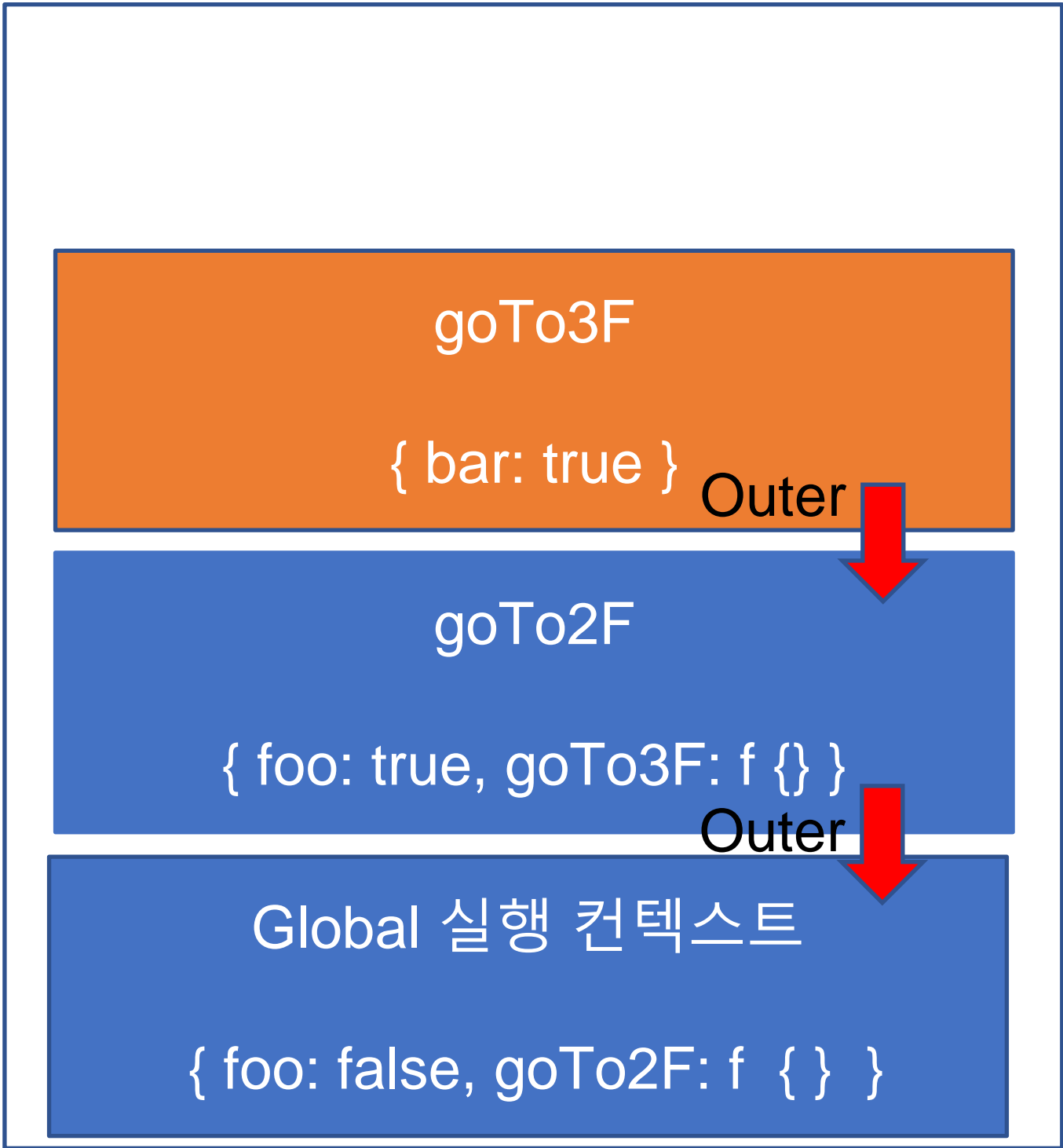
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
    console.log(foo);
  }

  goTo3F( );
}

goTo2F( );
```



Call stack

this, 실행 컨텍스트, 클로저



JS 엔진: foo 찾았다! 밖에 있었구만

☑ 실행 컨텍스트



```
let foo = false;

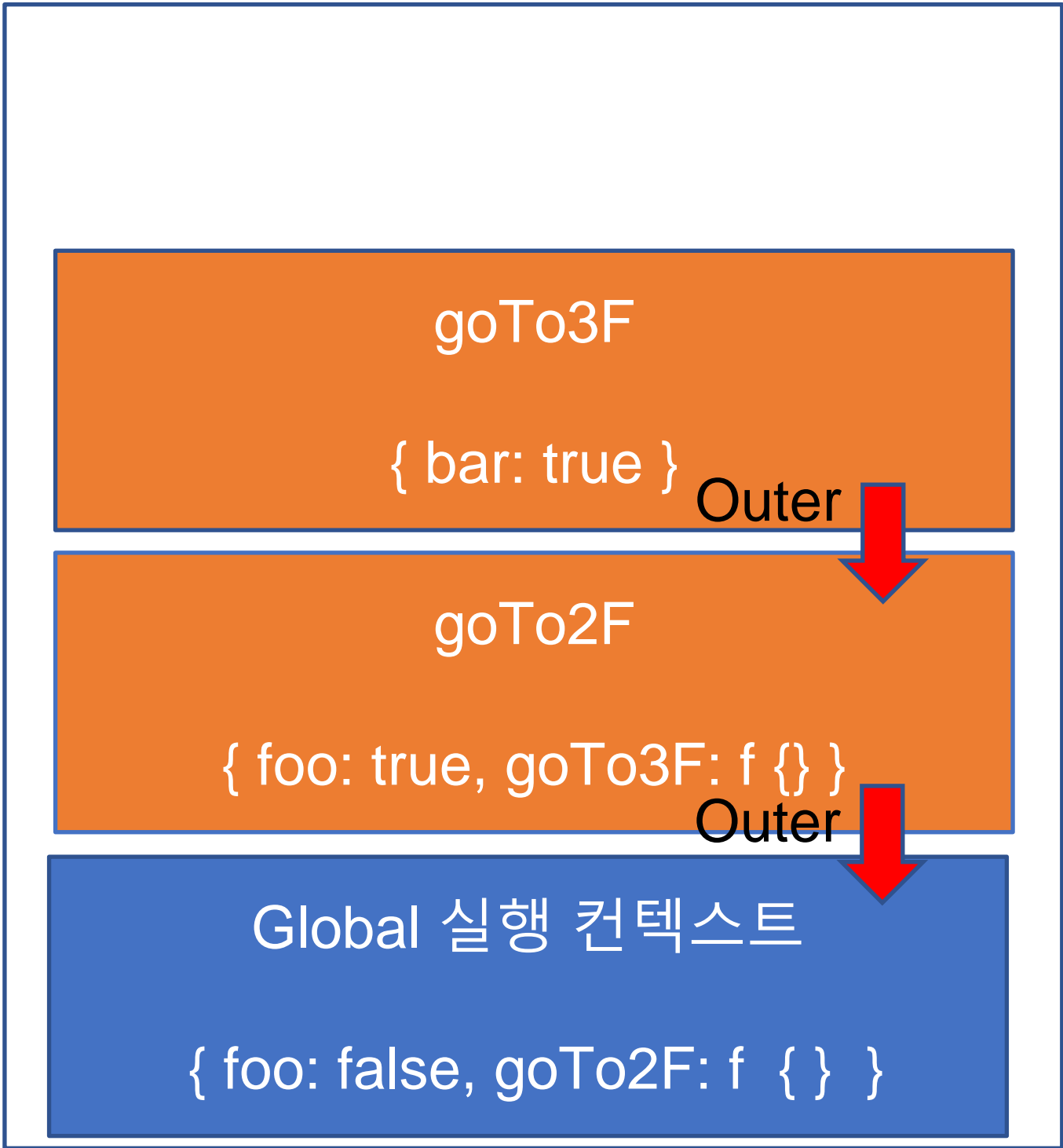
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
    console.log(foo);
  }

  goTo3F();
}

goTo2F();
```



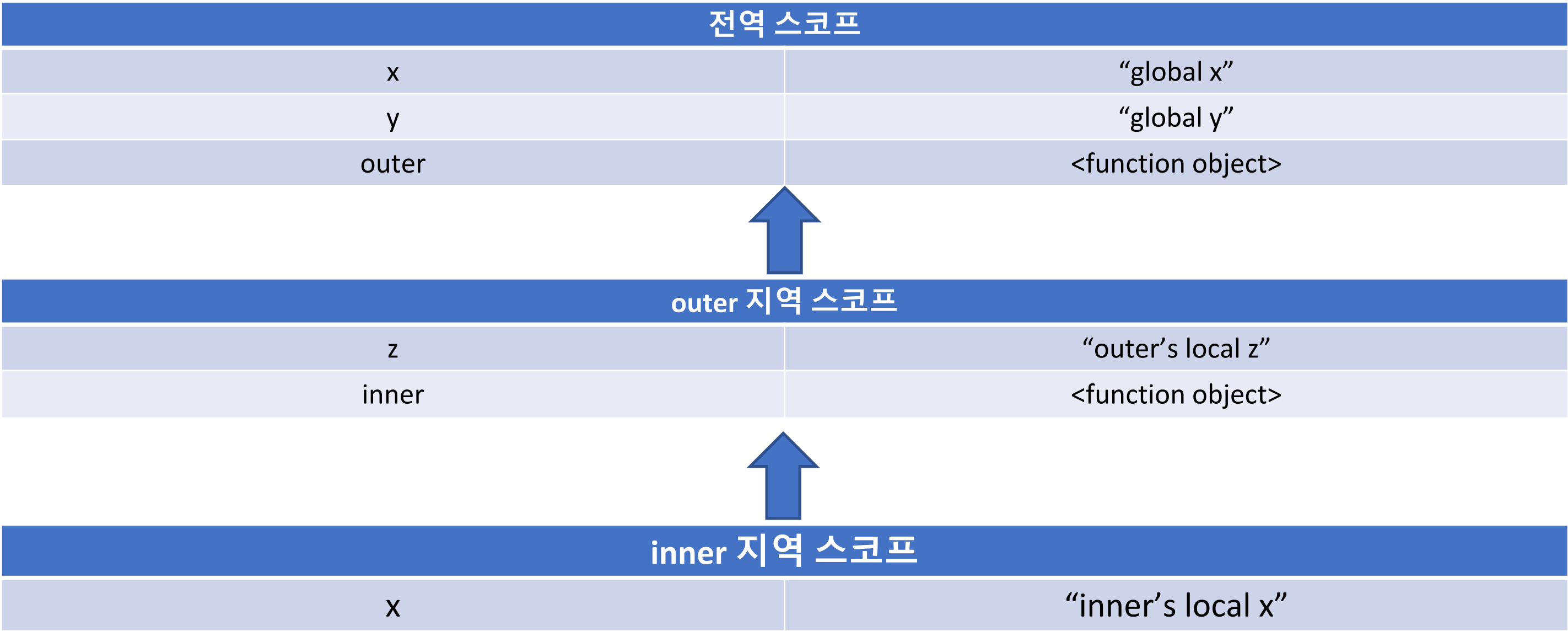
Call stack

변수 새도잉

동일한 식별자로 인해
상위 스코프에서 선언된 식별자의 값이
가려지는 현상

☑ 스코프

스코프 체인: 스코프가 계층적으로 연결된 것



this, 실행 컨텍스트, 클로저

☑ 실행 컨텍스트



```
let foo = false;

function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

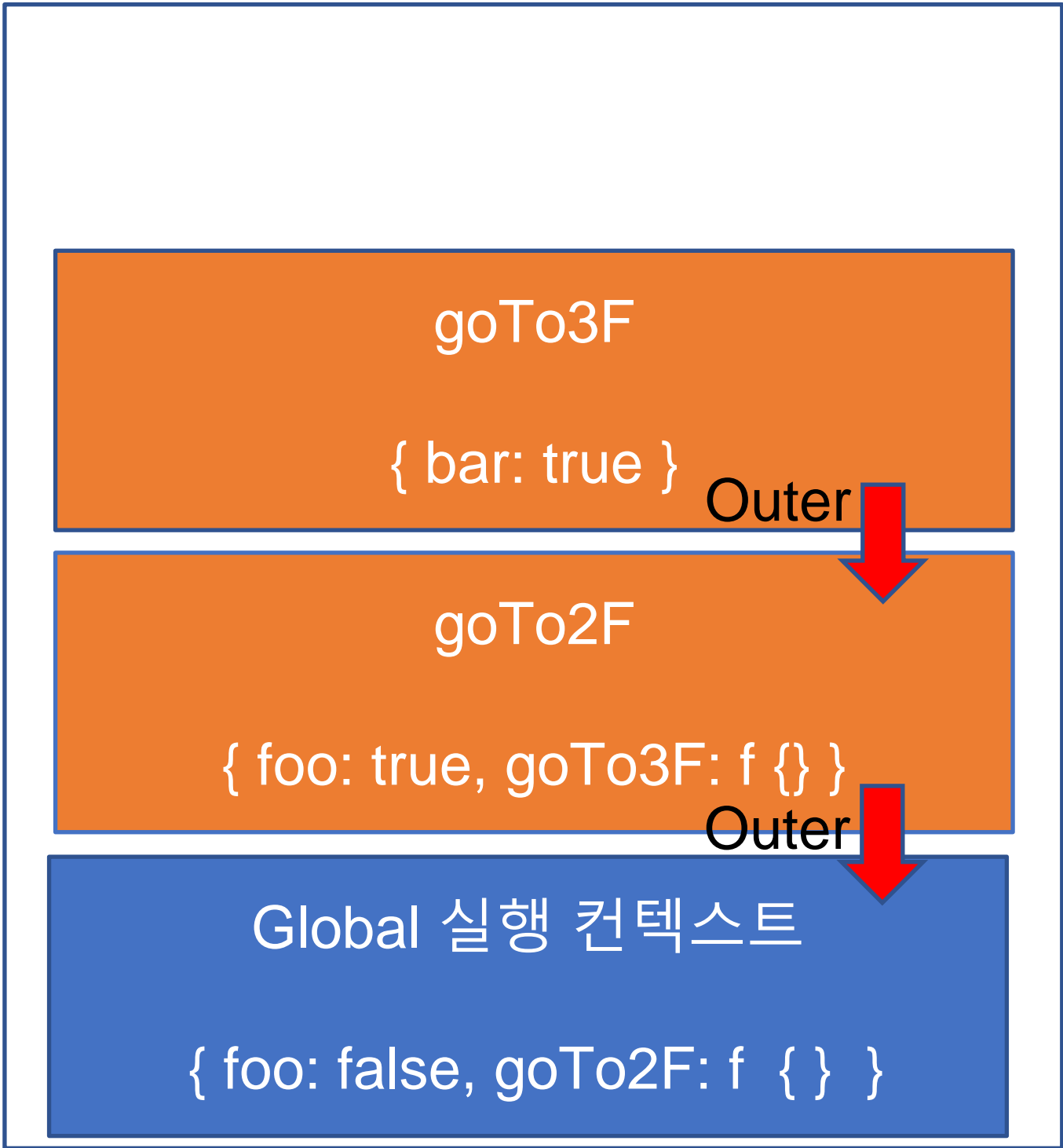
    console.log(bar);
    console.log(foo);
  }

  goTo3F( );
}

goTo2F( );
```




JS 엔진: 스코프 체이닝을 통해서
잘 찾을 수 있었어



Call stack

this, 실행 컨텍스트, 클로저

 JS 엔진: 실행 컨텍스트가 없었다면..끔찍!

☑ 실행 컨텍스트



```
let foo = false;

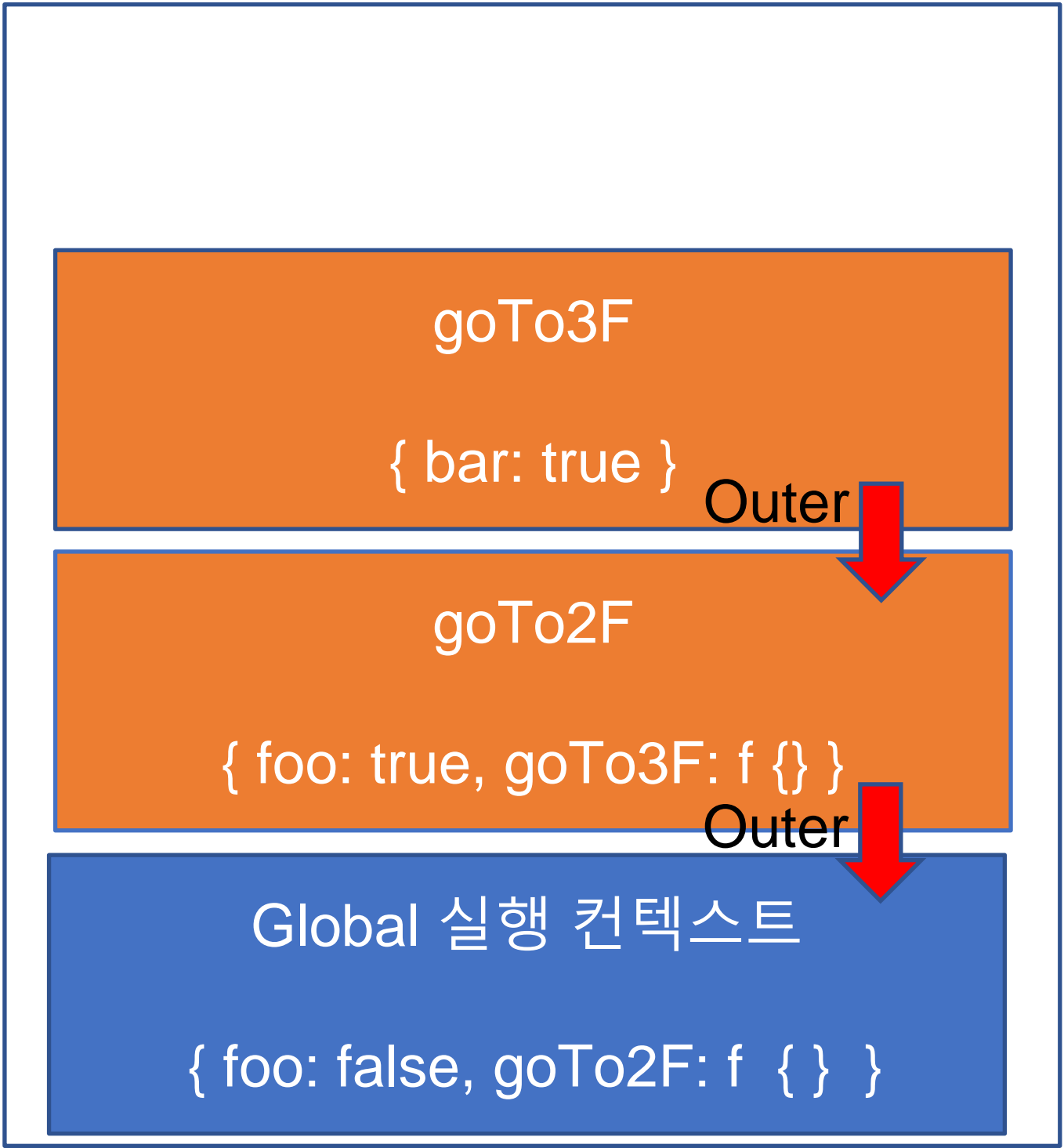
function goTo2F() {
  let foo = true;

  function goTo3F() {
    let bar = false;

    console.log(bar);
    console.log(foo);
  }

  goTo3F( );
}

goTo2F( );
```



Call stack

this, 실행 컨텍스트, 클로저

④ 실행 컨텍스트

실행 컨텍스트는

코드를 실행하는데 필요한 환경을 제공하는 객체이자

식별자 결정을 더욱 효율적으로 하기 위한 수단

03

클로저

this, 실행 컨텍스트, 클로저



JS 엔진: 활성화된 실행 컨텍스트는 전역!

☑ 클로저



```
const x = 1;


function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}
const foo = outSide();
foo();
```



Call stack

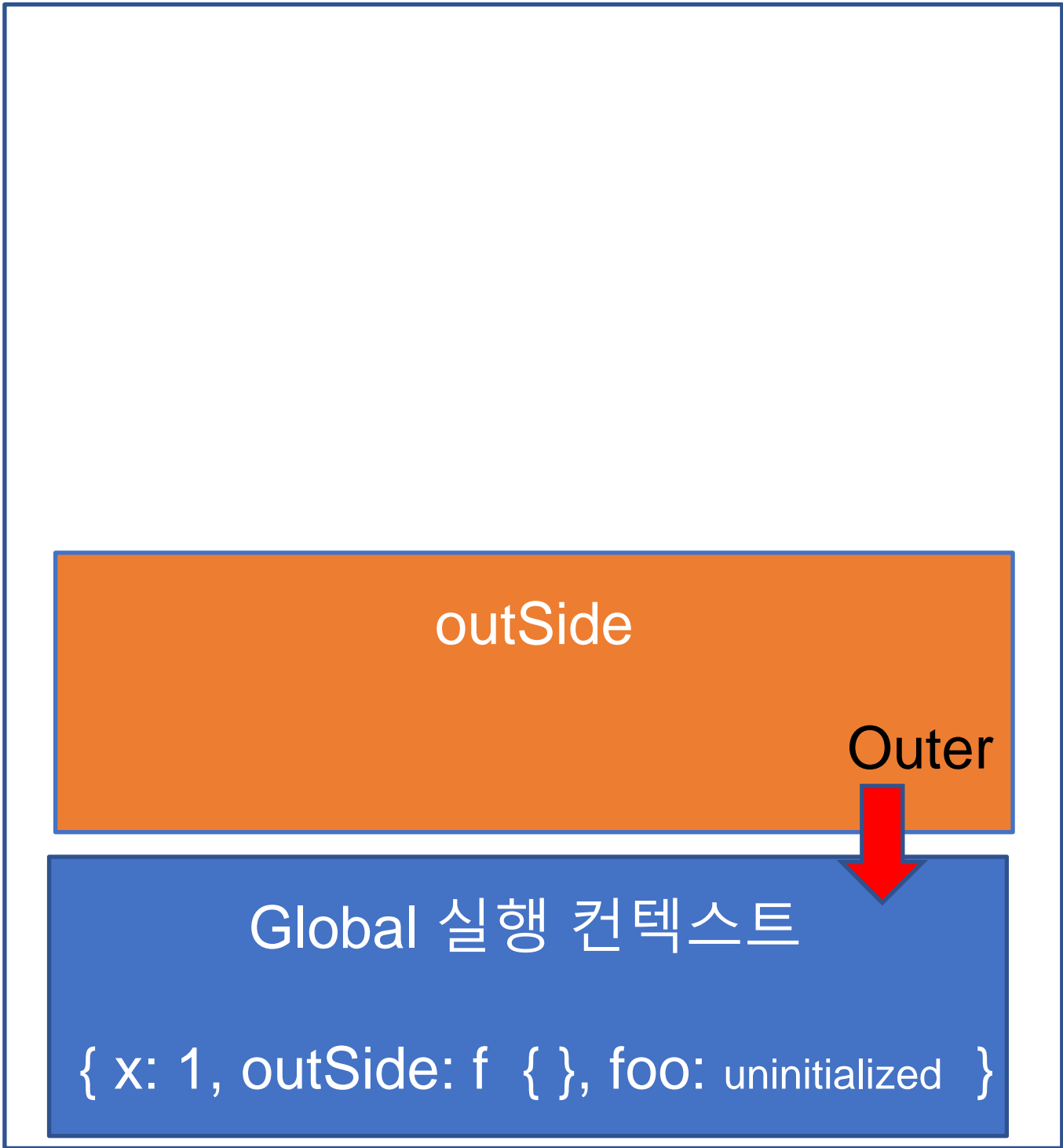
this, 실행 컨텍스트, 클로저

클로저

 JS 엔진: 활성화된 실행 컨텍스트는 outSide!


```
const x = 1;

function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}
const foo = outSide();
foo();
```



this, 실행 컨텍스트, 클로저

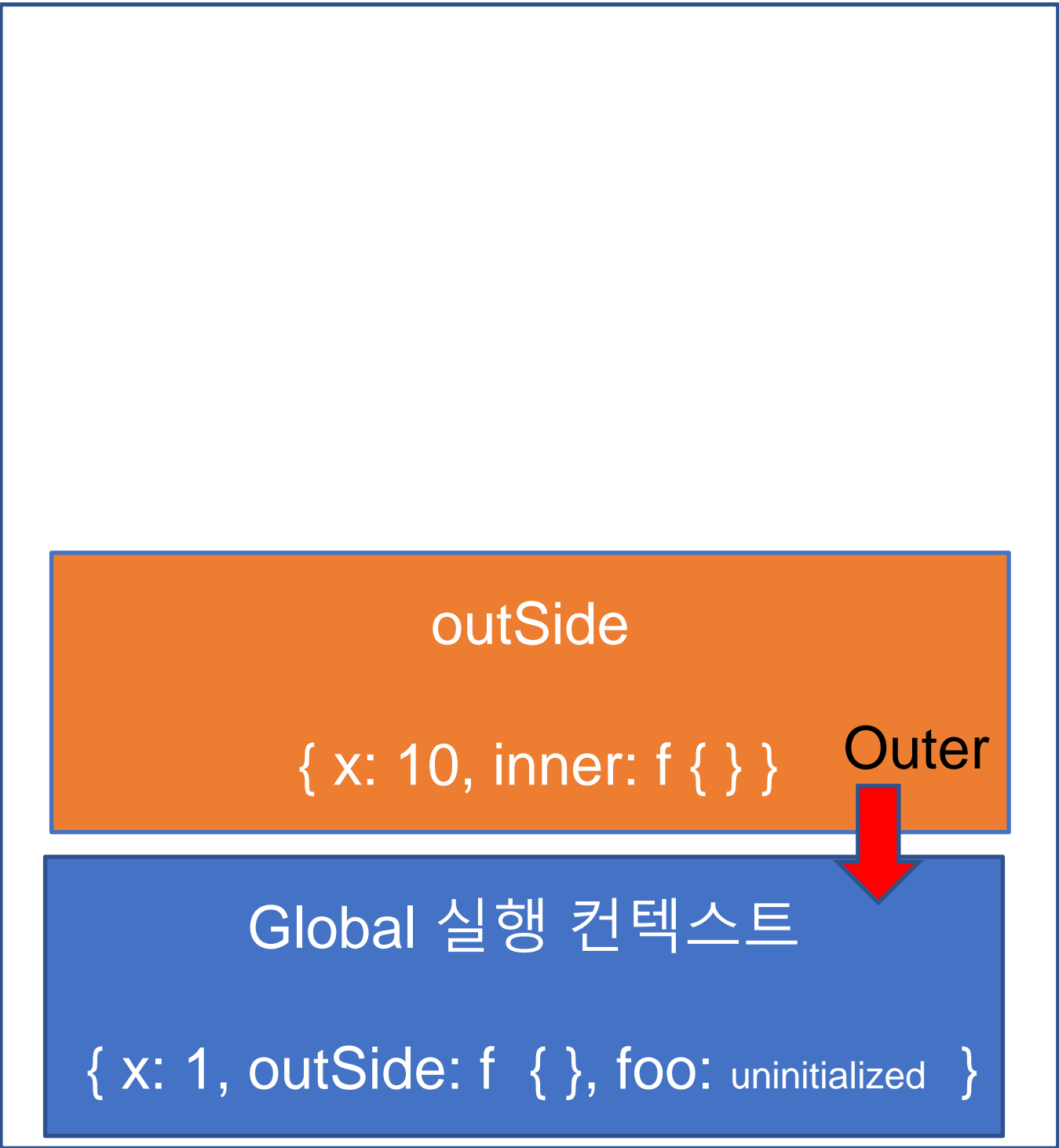
클로저

 JS 엔진: 활성화된 실행 컨텍스트는 outSide!

```
const x = 1;

function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}

const foo = outSide();
foo();
```



클로저



JS 엔진: outSide 실행 컨텍스트는 없애고 foo에 값을 할당하자!

```
const x = 1;

function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}
const foo = outSide();
foo();
```

Global 실행 컨텍스트

{ x: 1, outer: f {}, foo: f {} }

Call stack

this, 실행 컨텍스트, 클로저

클로저

 JS 엔진: foo를 실행시키자!

```
const x = 1;


function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}
const foo = outSide();
foo();
```

Global 실행 컨텍스트

{ x: 1, outer: f {}, foo: f {} }

Call stack

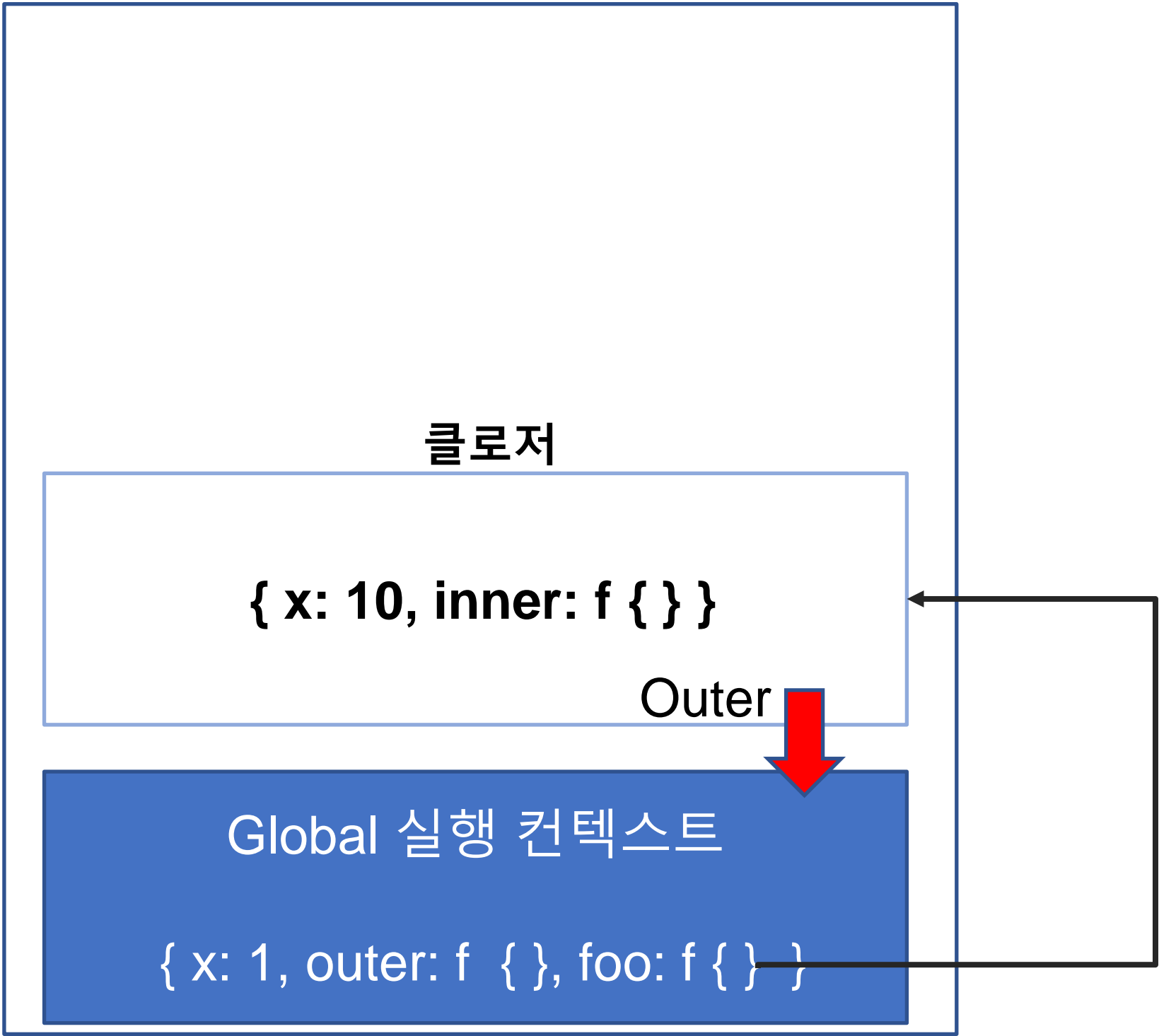
클로저

 JS 엔진: 클로저라서 아직 접근할 수 있군!

```
const x = 1;

function outSide() {
  const x = 10;
  const inner = function () {
    console.log(x);
  }
  return inner;
}

const foo = outSide();
foo();
```



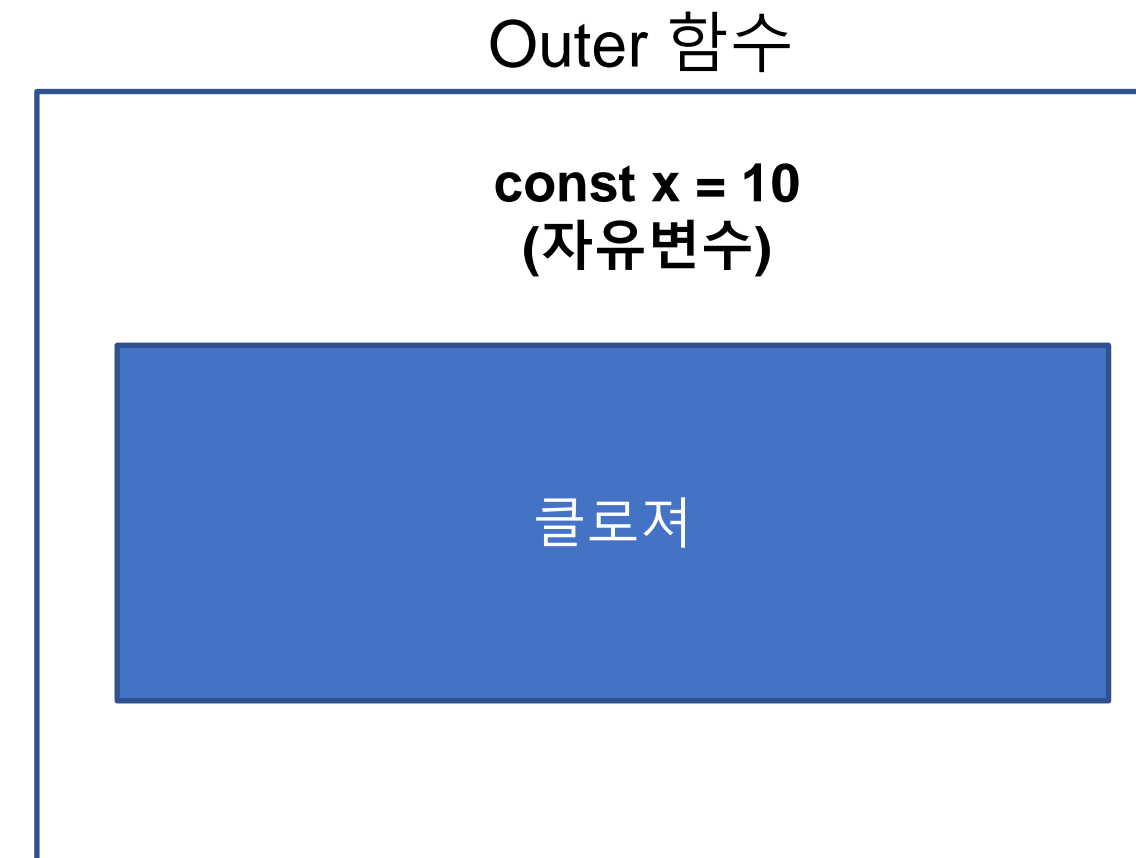
Call stack

this, 실행 컨텍스트, 클로저

☑ 클로저

한 중첩 함수가 상위 스코프의 식별자를 참조하고 있고
본인의 외부 함수보다 더 오래 살아있다면 그 중첩 함수는 **클로저**이다.

참고)
클로저에 의해 참조되는 변수를 자유 변수 라고 함



04

import, export

import, export

④ import

파일 별로 관리할 때, 파일 각각을 모듈이라고 부름

- export 키워드: 모듈 내보내기
- import 키워드: 모듈 가져오기

모듈 기능은 로컬에서 동작하지 않으므로, live server 같이 HTTP(S) 프로토콜을 지원하는 환경에서 써야함

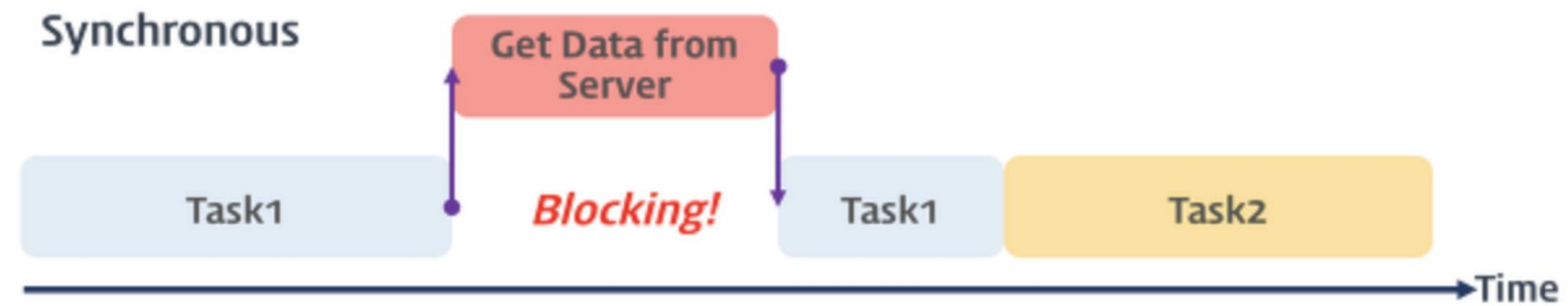
05

동기, 비동기

동기, 비동기

☑ 동기, 비동기

동기 처리 방식: 직렬적으로 일을 수행

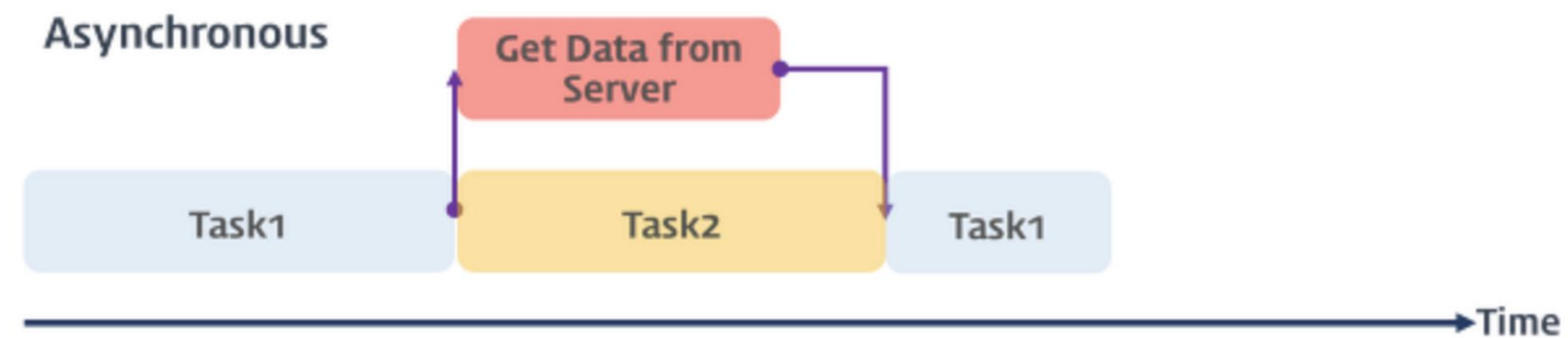


동기식 처리 모델(Synchronous processing model)

동기, 비동기

☑ 동기, 비동기

비동기 처리 방식: 병렬적으로 일을 수행



비동기식 처리 모델(Asynchronous processing model)

06

콜백지옥, promise 맛보기

동기, 비동기

☑ 콜백지옥, promise 맛보기

콜백 지옥



```
1  function hell(win) {  
2    // for listener purpose  
3    return function() {  
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {  
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {  
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {  
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {  
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {  
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {  
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {  
11                loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {  
12                 loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {  
13                  async.eachSeries(SCRIPTS, function(src, callback) {  
14                   loadScript(win, BASE_URL+src, callback);  
15                  });  
16                });  
17              });  
18            });  
19           });  
20          });  
21         });  
22        });  
23       });  
24      });  
25     };  
26   }  
}
```

👉 Promise

Promise는 콜백 패턴이 가진 단점을 보완한 또 다른 비동기 처리 패턴이자 객체

- 자바스크립트는 비동기 처리를 위한 하나의 패턴으로 콜백 함수를 사용함
- 콜백 함수는 가독성이 나쁘고 에러 처리가 번거로움



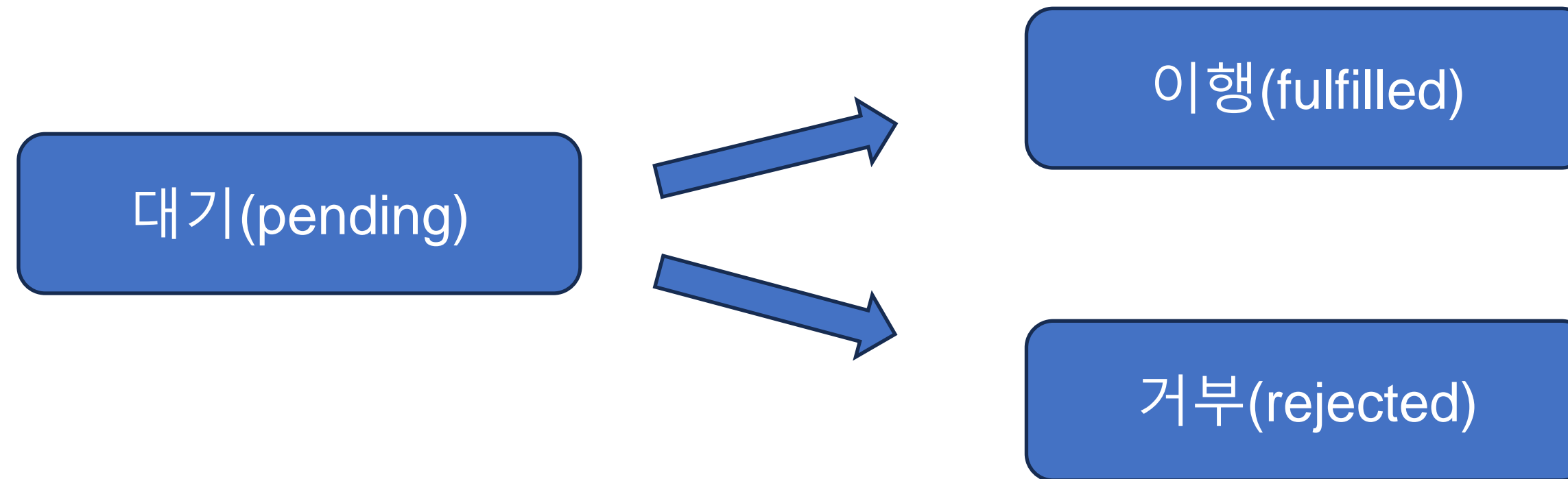
```
1 const myPromise = new Promise();
```



```
1 const myPromise = new Promise(() => {  
2  
3 });
```

☑ Promise

Promise 상태



☑ Promise

- Promise 문법
 - `new Promise((resolve, reject) => {
 if(/* 비동기 처리 성공 */) {
 resolve(...)
 } else { /* 비동기 처리 실패 */
 reject(...)
 }
})`

👍 Promise

- Promise 상태
 - 프로미스의 상태는 기본적으로 pending 상태이다. 이후 비동기 처리가 수행되면 결과에 따라 상태가 변경된다.

프로미스 상태	의미	해당 상태로 변경하기 위한 작업
pending(대기)	비동기 처리가 아직 수행되지 않은 상태	없음 (프로미스가 생성된 직후 기본 상태)
fulfilled(이행)	비동기 처리가 수행된 상태(성공)	resolve 함수 호출
rejected(거부)	비동기 처리가 수행된 상태(실패)	reject 함수 호출

자바스크립트 문법

☑ 참고

- 모던 자바스크립트 Deep Dive
- 짐코딩 CODING GYM
- 우아한 부트캠프
- 생활코딩