

# Execution Context



- ① Lexical Scope
- ② this 바인딩
- ③ Scope Chain(변수와 함수 찾기)
- ④ 호이스팅의 원인
- ⑤ 클로저

# Execution Context

진짜!  
모두를 위한  
자바스크립트



ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

자바스크립트 엔진이 스크립트 코드를 실행할 때

- ① 먼저 평가라는 것을 합니다.  
평가를 하면서 실행에 필요한 내용을  
생성하며 코드 실행을 준비한다.
- ② 이때 자바스크립트 엔진이 생성하는 것이  
실행 컨텍스트(Execution Context)
- ③ 즉, 실행컨텍스트는 자바스크립트 코드가  
실행되는 환경으로 생각할 수 있다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

... ExecutionContext

```
const ExecutionContext = {  
  Lexical_Environment: {  
    Object_Environment_Record: {  
      //...  
    },  
    Outer_Lexical_Environment_Reference: {  
      //...  
    },  
  },  
  Varialbe_Environment: {  
    // ...  
  },  
  ThisBinding: null,  
}
```

## Lexical & Variable Environment

- ✓ 변수 식별자와 변수 값을 관리한다.
- ✓ 즉, 변수를 위한 저장소다
- ✓ Scope를 관리한다. Lexical Envitonment 이 실습에서 보게될 [[Scopes]] 슬롯을 관리한다
- ✓ [[Scopes]] 슬롯 있어 Scope Chaining이 가능하다.
- ✓ with구문처럼 특정 경우를 제외하곤 Lexical과 Variable 환경이 서로 같기 때문에 이후 설명에서는 Lexical로 설명 한다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트



ExecutionContext - ex03

```
const ExecutionContext = {  
  Lexical_Environment: {  
    // ...  
  },  
  Varialbe_Environment: {  
    // ...  
  },  
  ThisBinding: null,  
};
```

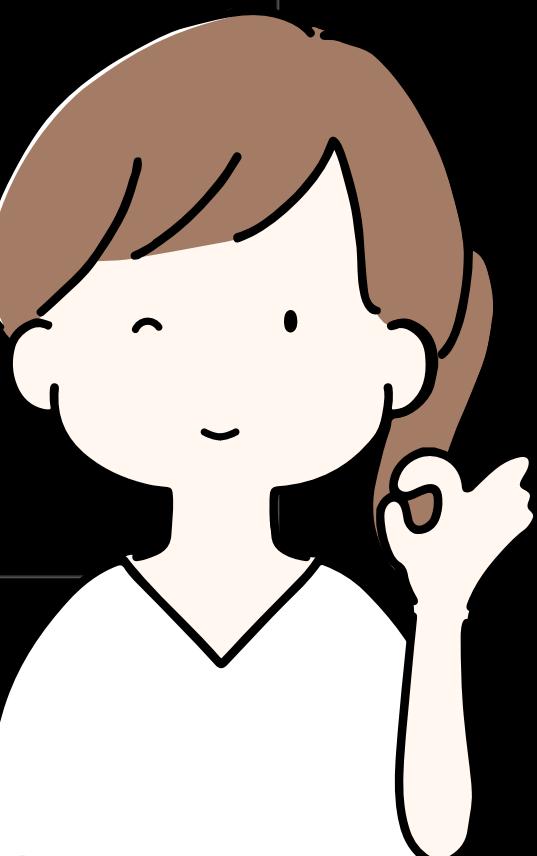
## ThisBinding

- ✓ 현재 실행 컨텍스트 내에서 **this** 키워드가 가리키는 객체
- ✓ 전역 실행 컨텍스트는 **globalThis**를 갖고 함수 실행 컨텍스트는 함수가 어떻게 호출 되었는지에 따라 달라진다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

```
...  
const ExecutionContext = {  
    Lexical_Environment: {  
        Object_Environment_Record: {  
            [[local]]: [  
                // 변수, 함수  
                // 만약 함수에 내부 함수가 있다면 내부 함수도  
            ]  
        },  
        Outer_Lexical_Environment_Reference: {  
            // local을 벗어난 변수와 함수를 찾기 위한 상위 스코프  
            // 찾는 순서는 위에서 아래로  
            [[Scopes]]: [  
                Closure,  
                Script,  
                Global  
            ]  
        },  
        this  
    }  
}
```



# Execution Context

```
...  
const ExecutionContext = {  
    Lexical_Environment: {  
        Object_Environment_Record: {  
            [[local]]: [  
                // 변수, 함수  
                // 만약 함수에 내부 함수가 있다면 내부 함수도  
            ]  
        },  
        Outer_Lexical_Environment_Reference: {  
            // local을 벗어난 변수와 함수를 찾기 위한 상위 스코프  
            // 찾는 순서는 위에서 아래로  
            [[Scopes]]: [  
                Closure,  
                Script,  
                Global  
            ]  
        }  
    },  
    this  
}
```

진짜!  
모두를 위한  
자바스크립트

## 호이스팅이 일어나는 이유

- ① 현재 실행 컨텍스트를 만드는 과정 중 현재 실행 컨텍스트의 local 스코프에 있는 지역 변수의 정보를 Object Environment Record에 담기 때문에 호이스팅이 발생한다.
- ② 즉, 같은 스코프에 있다면 변수의 선언 위치는 무시되는 것이다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

## Lexical Scope

```
...  
Lexical Scope  
  
let a = 10;  
let b = 20;  
  
function test1() {  
    console.log(a);  
    console.log(b);  
    console.log(c);  
}  
  
let c = 30;  
  
test1();
```

test1함수가 c 변수를  
참조할 수 있나?



# Execution Context

진짜!  
모두를 위한  
자바스크립트

## Lexical Scope

• • • Lexical Scope

```
let a = 10;  
let b = 20;  
  
function test1() {  
    console.log(a);  
    console.log(b);  
    console.log(c);  
}  
  
test1();  
  
let c = 30;
```

이렇게 하면  
c를 참조할 수 있을까?



# Execution Context

진짜!  
모두를 위한  
자바스크립트

## Lexical Scope

...

Lexical Scope

```
let a = 10;  
let b = 20;  
  
function test1() {  
    console.log(a);  
    console.log(b);  
    console.log(c);  
}  
  
test1();  
  
var c = 30;
```

이렇게 하면  
c를 참조할 수 있을까?



# Execution Context ⚙

진짜!  
모두를 위한  
자바스크립트

c 변수를 다른 Lexical Scope 갖도록 선언하면 된다.

```
...  
Lexical Scope  
  
let a = 10;  
let b = 20;  
  
function test1() {  
  console.log(a);  
  console.log(b);  
  console.log(c); // ReferenceError: c is not defined  
}  
  
function test2() {  
  let c = 30;  
}  
  
test1();
```

```
...  
Lexical Scope  
  
let a = 10;  
let b = 20;  
  
{  
  let c = 30; // 블록 스코프 내에서 선언  
}  
  
function test1() {  
  console.log(a);  
  console.log(b);  
  console.log(c); // ReferenceError: c is not defined  
}  
  
test1();
```

c변수를 참조하지  
못하게 하려면  
어떻게 해야 할까?



# Execution Context

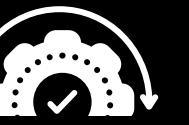


진짜!  
모두를 위한  
자바스크립트

- 1 전역 실행 컨텍스트(Global Execution Context)
- 2 함수 실행 컨텍스트(Function Execution Context)
- 3 Eval 함수 실행 컨텍스트(Eval Function Execution Context)

즐거운 코딩 경험  
⟨CODINGMAX⟩

# Execution Context



진짜! 모두를 위한  
자바스크립트



ExecutionContext - ex04

```
const GlobalExecutionContext = {  
    LexicalEnvironment: {  
        // ...  
        // 상위 스코프는 없다.  
        [[Scopes]]: null  
    },  
    ThisBinding: globalThis  
}
```

## 전역 실행 컨텍스트

- ✓ 가장 최상위 실행 컨텍스트
- ✓ 자바스크립트 코드 최초로 실행될 때, 만들어진다.
- ✓ Lexical 환경에서 Global 스코프를 관리한다.
- ✓ 따라서 특정 함수나 블럭이 아닌 Global 스코프에 정의된 var 변수 및 함수 식별자(함수 표현식으로 할당한)는 Global 스코프에 존재한다.
- ✓ let, const 변수는 script 스코프에 존재한다.
- ✓ this는 globalThis를 가리킨다. 브라우저에서는 전역 실행 컨텍스트의 this가 window 객체이다.

즐거운 코딩 경험  
〈CODINGMAX/〉

# Execution Context

진짜!  
모두를 위한  
자바스크립트

함수 실행 컨텍스트(ex04.js)

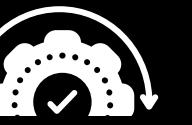
```
const FunctionExecutionContext = {
  LexicalEnvironment: {
    local_variables: {
      // ...
    },
    nested_functions: {
      // ...
    },
    arguments: {
      // ...
    },
    [[Scopes]]: {
      // 상위 스코프
      // 함수를 호출한 Lexical 환경
      // [[Scopes]] === [[Parent Lexical Envitonment]]
    }
  },
  ThisBinding: getThisBinding()
}

function getThisBinding() {
  switch (함수종류) {
    case '일반함수':
      return strictMode ? undefined : globalThis;
    case '메서드':
      return '메서드를 호출한 객체';
    case '생성자함수':
      return '새로 생성된 객체';
    case 'apply/bind/call':
      return '개발자가 명시적으로 this 설정';
    case '화살표함수':
      return '상위 Lexical Context에서 this 상속';
  }
}
```

## 함수 실행 컨텍스트

- ✓ 함수가 호출되어 실행될 때, 함수 실행 컨텍스트가 매 호출마다 새로 생성된다.
- ✓ 지역변수, 함수 파라미터 및 인자값, 내장 함수 등을 담고 있다.
- ✓ [[Scopes]]로 함수가 선언된 상위 Lexical 환경을 참조하고 있다. [[Scopes]]가 있어 Scope Chaining이 가능하다.
- ✓ ThisBinding은 함수 종류에 따라 다르게 바인딩 된다.(실행 컨텍스트 Part 2에서 다룸)

# Execution Context



진짜!  
모두를 위한  
자바스크립트



함수 실행 컨텍스트(ex04\_1.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 함수 실행 컨텍스트는 함수가 호출되어 실행 될 때마다  
 * 새로 만들어진다.  
 */  
function a_func(a, b) {  
    console.log(a, b);  
}  
  
console.log('START');  
a_func(10, 20);
```

# Execution Context

진짜!  
모두를 위한  
자바스크립트



함수 실행 컨텍스트(ex04\_2.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 상위(또는 Parent) 스코프란 무엇인가?  
 */  
  
function a_func1() {  
    const n = 10;  
    a_func2();  
    console.log(`n is ${n}`);  
}  
  
function a_func2() {  
    console.log(`a_func2's name is ${a_func2.name}`);  
}  
  
console.log('START');  
a_func1();
```

# Execution Context

진짜!  
모두를 위한  
자바스크립트



함수 실행 컨텍스트(ex04\_3.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 상위(또는 Parent) 스코프란 무엇인가?  
 */  
  
// 상위스코프 캡쳐링이 없을 때,  
function a_func1() {  
    const n = 10;  
    const a_func2 = function () {  
        console.log(`a_func2's name is ${a_func2.name}`);  
    }  
    a_func2();  
    console.log(`n is ${n}`);  
}  
  
console.log('START');  
a_func1();
```



함수 실행 컨텍스트(ex04\_3.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 상위(또는 Parent) 스코프란 무엇인가?  
 */  
  
// 상위스코프 캡쳐링이 있을 때,  
function a_func1() {  
    const n = 10;  
    const a_func2 = function () {  
        console.log(`a_func2's name is ${a_func2.name}`);  
        console.log(`n is ${n}`);  
    }  
    a_func2();  
    console.log(`n is ${n}`);  
}  
  
console.log('START');  
a_func1();
```

# Execution Context



진짜!  
모두를 위한  
자바스크립트

● ● ●

함수 실행 컨텍스트(ex04\_4.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 다시 보는 함수스코프와 블록스코프  
 */  
  
// 함수 실행 컨텍스트는 함수가 실행될 때 생성된다.  
// 따라서 1초가 지난 뒤에 setTimeout의  
// 핸들러 함수가 호출되기 때문에 i 는 4 가 3 번 출력되는 것이다.  
function a_counter() {  
    for(var i = 1; i <= 3; i++) {  
        setTimeout(function () {  
            console.log(i);  
        }, 1000);  
    }  
    console.log('for 루프 종료', i);  
}  
a_counter();
```

● ● ●

함수 실행 컨텍스트(ex04\_4.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 다시 보는 함수스코프와 블록스코프  
 */  
  
// 블록스코프  
function a_counter() {  
    for(let i = 1; i <= 3; i++) {  
        setTimeout(function () {  
            console.log(i);  
        }, 1000);  
    }  
    // 블록스코프이므로 여기서 i에 접근할 수 없다.  
    console.log('for 루프 종료');  
}  
a_counter();
```

● ● ●

함수 실행 컨텍스트(ex04\_4.js)

```
/**  
 * 함수 실행 컨텍스트  
 * 다시 보는 함수스코프와 블록스코프  
 */  
  
// 즉시실행함수  
// 즉, 함수 실행 컨텍스트를 바로 만들어줘서  
// 루프 변수 i 값을 즉시함수 스코프로 복사해 두는 것이다.  
function a_counter() {  
    for(var i = 1; i <= 3; i++) {  
        (function () {  
            var clone = i;  
            setTimeout(function () {  
                console.log(clone);  
            }, 1000);  
        })();  
    }  
    console.log('for 루프 종료', i);  
}  
a_counter();
```

즐거운 코딩 경험  
⟨CODINGMAX⟩

# Execution Context

진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex05

```
const EvalExecutionContext = {  
    LexicalEnvironment: {  
        //...  
        [[Scopes]]: stricMode ? 'eval 자체만 접근 가능한 격리 스코프'  
        : 'eval 내외부 컨텍스트에 접근할 수 있는 스코프'  
    },  
    ThisBinding: '호출한 컨텍스트의 this값을 상속'  
}
```

## Eval 실행 컨텍스트

- ✓ eval() 함수는 문자열로 된 자바스크립트 코드를 실행한다.
- ✓ eval() 함수가 호출될 때, Eval 실행 컨텍스트가 생성된다.
- ✓ Eval 실행 컨텍스트는 자신만의 Scope Chain과 Lexical 환경을 갖는다.
- ✓ ThisBinding은 호출한 컨텍스트의 this값을 상속 한다.
- ✓ eval함수는 보안상 위험하기 때문에 사용하지 않는다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex06

```
// strict 모드가 아니면 외부에서 eval 실행 컨텍스트에 접근할 수 있다.
```

```
function exampleEval() {  
    var x = 10;  
    // eval 내에서 변수 y 선언  
    eval('var y = 20;');
```

```
// 10 출력, x는 함수 스코프 내에서 선언되었으므로 접근 가능  
console.log(x);
```

```
// 20 출력, y는 eval 내에서 선언되었으나 비엄격 모드에서는 함수 스코프에서 접근 가능  
console.log(y);
```

```
}
```

```
exampleEval();
```

## Eval 실행 컨텍스트

- ✓ eval() 함수는 문자열로 된 자바스크립트 코드를 실행한다.
- ✓ eval() 함수가 호출될 때, Eval 실행 컨텍스트가 생성된다.
- ✓ Eval 실행 컨텍스트는 자신만의 Scope Chain과 Lexical 환경을 갖는다.
- ✓ ThisBinding은 호출한 컨텍스트이 this값을 상속 한다.
- ✓ eval함수는 보안상 위험하기 때문에 사용하지 않는다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex07

```
// strict 모드이면 외부에서 eval 실행 컨텍스트에 접근할 수 없다.  
function exampleEval() {  
    'use strict'; // strict 모드 활성화  
  
    var x = 10;  
    // eval 내에서 변수 y 선언  
    eval('var y = 20;');  
  
    console.log(x);  
    console.log(y); // ReferenceError 발생, y는 eval의 렉시컬 스코프 내에서만 존재합니다  
}  
  
exampleEval();
```

## Eval 실행 컨텍스트

- ✓ eval() 함수는 문자열로 된 자바스크립트 코드를 실행한다.
- ✓ eval() 함수가 호출될 때, Eval 실행 컨텍스트가 생성된다.
- ✓ Eval 실행 컨텍스트는 자신만의 Scope Chain과 Lexical 환경을 갖는다.
- ✓ ThisBinding은 호출한 컨텍스트이 this값을 상속 한다.
- ✓ eval함수는 보안상 위험하기 때문에 사용하지 않는다.

# Execution Context

진짜!  
모두를 위한  
자바스크립트

Eval 실행 컨텍스트

```
// strict 모드에 상관 없이 eval 실행 컨텍스트는 외부 스코프에 접근할 수 있다.  
const x = 5;  
function test() {  
    const y = 10;  
    eval(`console.log(x + y);`);  
    console.log('end of test');  
}  
test(); // 15  
  
//-----  
  
'use strict'; // strict 모드 활성화  
const x = 5;  
function test() {  
    const y = 10;  
    eval(`console.log(x + y);`);  
    console.log('end of test');  
}  
test(); // 15  
  
//-----  
  
const x = 5;  
function test() {  
    'use strict'; // strict 모드 활성화  
    const y = 10;  
    eval(`console.log(x + y);`);  
    console.log('end of test');  
}  
test(); // 15
```

## Eval 실행 컨텍스트

- ✓ eval() 함수는 문자열로 된 자바스크립트 코드를 실행한다.
- ✓ eval() 함수가 호출될 때, Eval 실행 컨텍스트가 생성된다.
- ✓ Eval 실행 컨텍스트는 자신만의 Scope Chain과 Lexical 환경을 갖는다.
- ✓ ThisBinding은 호출한 컨텍스트의 this값을 상속 한다.
- ✓ eval함수는 보안상 위험하기 때문에 사용하지 않는다.

즐거운 코딩 경험  
〈CODINGMAX/〉

This  
Binding



# THIS → Binding

진짜!  
모두를 위한  
자바스크립트

```
function getThisBinding(함수종류) {
    switch (함수종류) {
        case '일반함수':
            return strictMode ? undefined : globalThis;
        case '메서드':
            return '메서드를 호출한 객체';
        case '생성자함수':
            return '새로 생성된 객체';
        case 'apply/bind/call':
            return '개발자가 명시적으로 this 설정';
        case '화살표함수':
            return '상위 Lexical Context에서 this 상속';
    }
}
```

# THIS → Binding

진짜!  
모두를 위한  
자바스크립트

- ① 렉시컬 스코프와 렉시컬 `this` 가 결정되는 방법 또는 시기를 혼동할 수 있다.
- ② 렉시컬 스코프는 함수가 선언된 위치를 기준으로 결정 된다.  
    시기를 굳이 말하자면 코드를 작성할 때 렉시컬 스코프가 결정된다.  
    ⇒ 정적
- ③ 렉시컬 `this`는 함수를 호출할 때 결정 된다.  
    즉, 함수 실행 컨텍스트가 만들어지면서 렉시컬 환경의 `this` 바인딩을 통해 결정 된다.  
    ⇒ 동적



# Execution Context Stack

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

## 실행 컨텍스트 스택

- ✓ 자바스크립트 엔진은 함수 호출 스택 (CallStack)을 관리한다.
- ✓ 함수가 호출될 때 생성되는 실행 컨텍스트를 함수 호출 스택에 LIFO(Last-In, First-Out) 순서로 추가하고 삭제한다.
- ✓ 가장 마지막으로 호출된 함수의 실행 컨텍스트가 호출 스택의 최상단에 추가되고 해당 함수가 종료되면 해당 함수의 실행 컨텍스트가 호출 스택에서 제거된다.

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

함수 호출 스택

console

anonymous

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

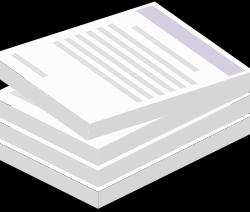
함수 호출 스택

console

printABC

anonymous

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

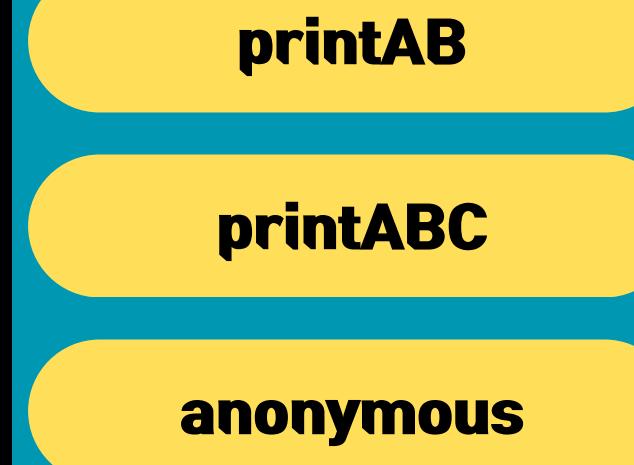
function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

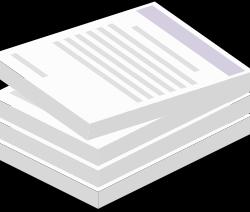
printABC();
```

함수 호출 스택



console

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

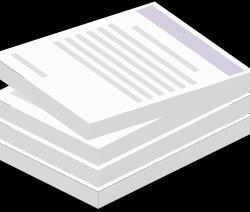
printABC();
```

함수 호출 스택

printA  
printAB  
printABC  
anonymous

console

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

함수 호출 스택

printAB

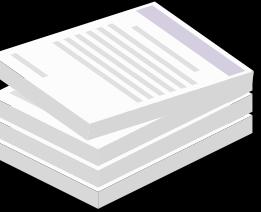
printABC

anonymous

console

a

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

함수 호출 스택

console

printABC

anonymous

a  
b

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

함수 호출 스택

console

anonymous

a  
b  
c

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

...

ExecutionContext - ex01

```
var a = 'This is var variable a';
let b = 'This is let variable b';
const c = 'This is const constant c';

function printA() {
    console.log('a', a);
}

function printAB() {
    printA();
    console.log('b', b);
}

function printABC() {
    printAB();
    console.log('c', c);
}

printABC();
```

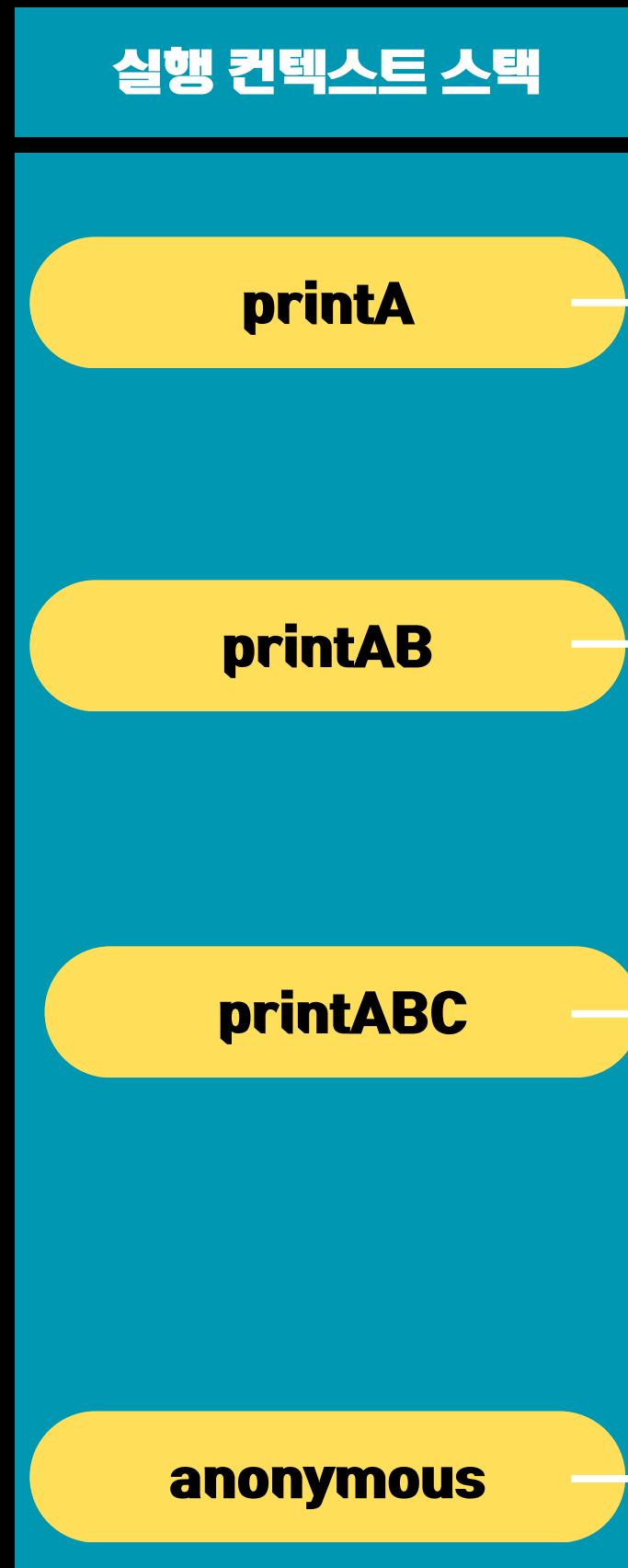
함수 호출 스택

console

a  
b  
c

# Execution Context Stack

진짜! 모두를 위한  
자바스크립트



```
const printAExecutionContext = {
  LexicalEnvironment: {
    local_variables: {
      arguments,
      caller: [function printAB],
      name: 'printA'
    },
    [[Scopes]]: [
      { local: this(> globalThis) },
      { script: b, c },
      { globalThis: window }
    ]
  },
  ThisBinding: globalThis
}, ThisBinding: globalThis

const printABExecutionContext = {
  LexicalEnvironment: {
    local_variables: {
      arguments,
      caller: [function printABC],
      name: 'printAB'
    },
    [[Scopes]]: [
      { local: this(> globalThis) },
      { script: b, c },
      { globalThis: window }
    ]
  },
  ThisBinding: globalThis
}, ThisBinding: globalThis

const printABCExecutionContext = {
  LexicalEnvironment: {
    local_variables: {
      arguments,
      caller: null,
      name: 'printABC'
    },
    [[Scopes]]: [
      { local: this(> globalThis) },
      { script: b, c },
      { globalThis: window }
    ]
  },
  ThisBinding: globalThis
}, ThisBinding: globalThis

const GlobalExecutionContext = {
  LexicalEnvironment: {},
  local_variables: {
    printA: [function object],
    printAB: [function object],
    printABC: [function object],
  },
  [[Scopes]]: null
}, ThisBinding: globalThis
}, ThisBinding: globalThis
```

# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

**fib(n) = fib(n-1) + fib(n-2)**

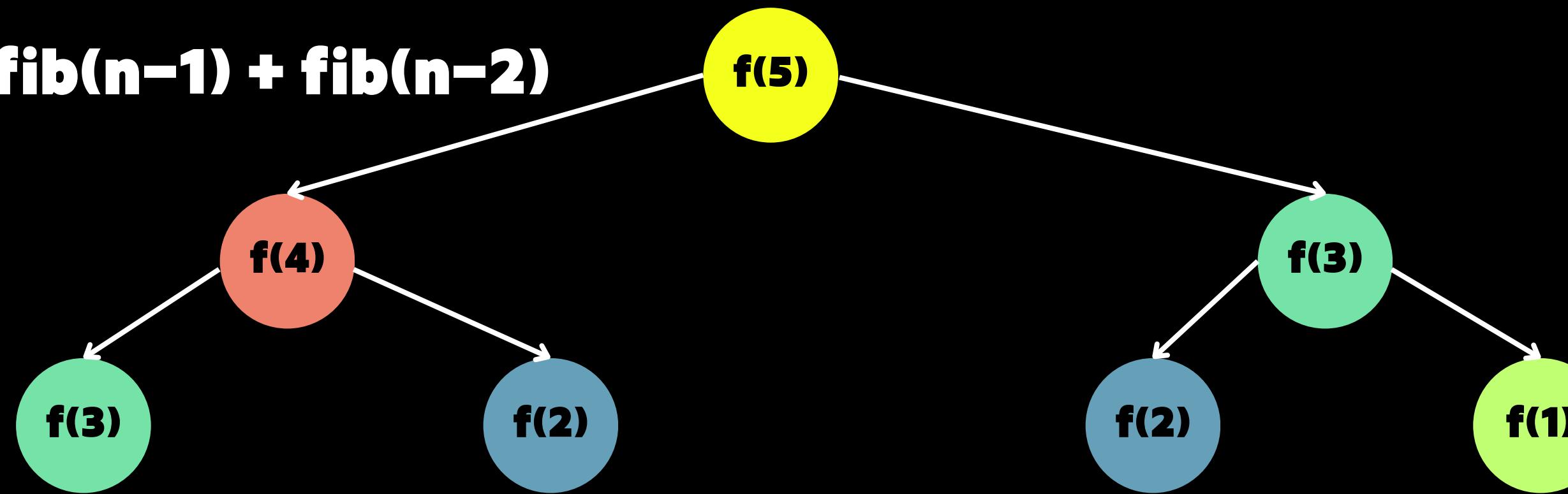
f(5)

# Execution Context Stack

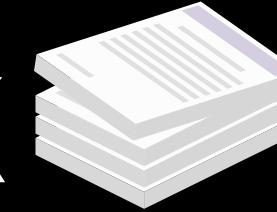


진짜!  
모두를 위한  
자바스크립트

**fib(n) = fib(n-1) + fib(n-2)**

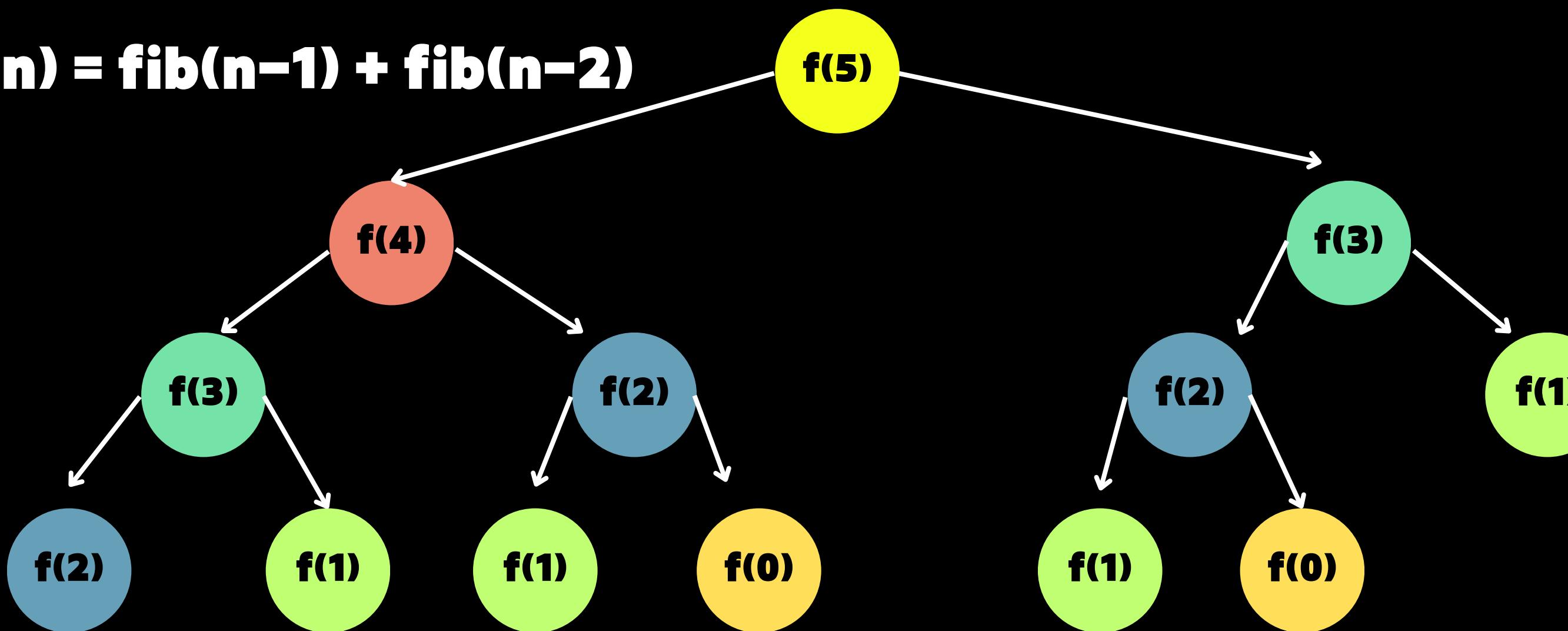


# Execution Context Stack



진짜!  
모두를 위한  
자바스크립트

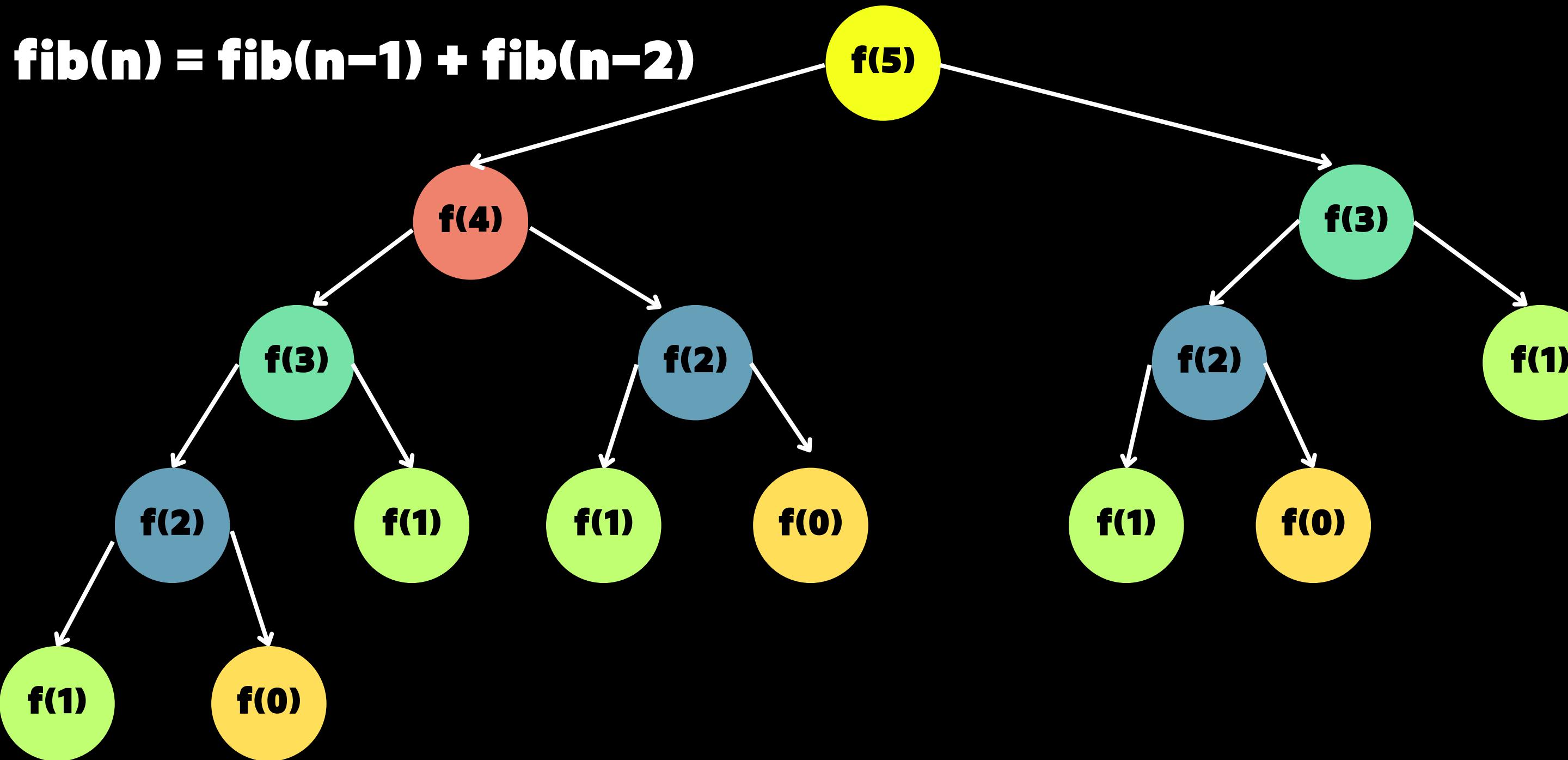
$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$



# Execution Context Stack

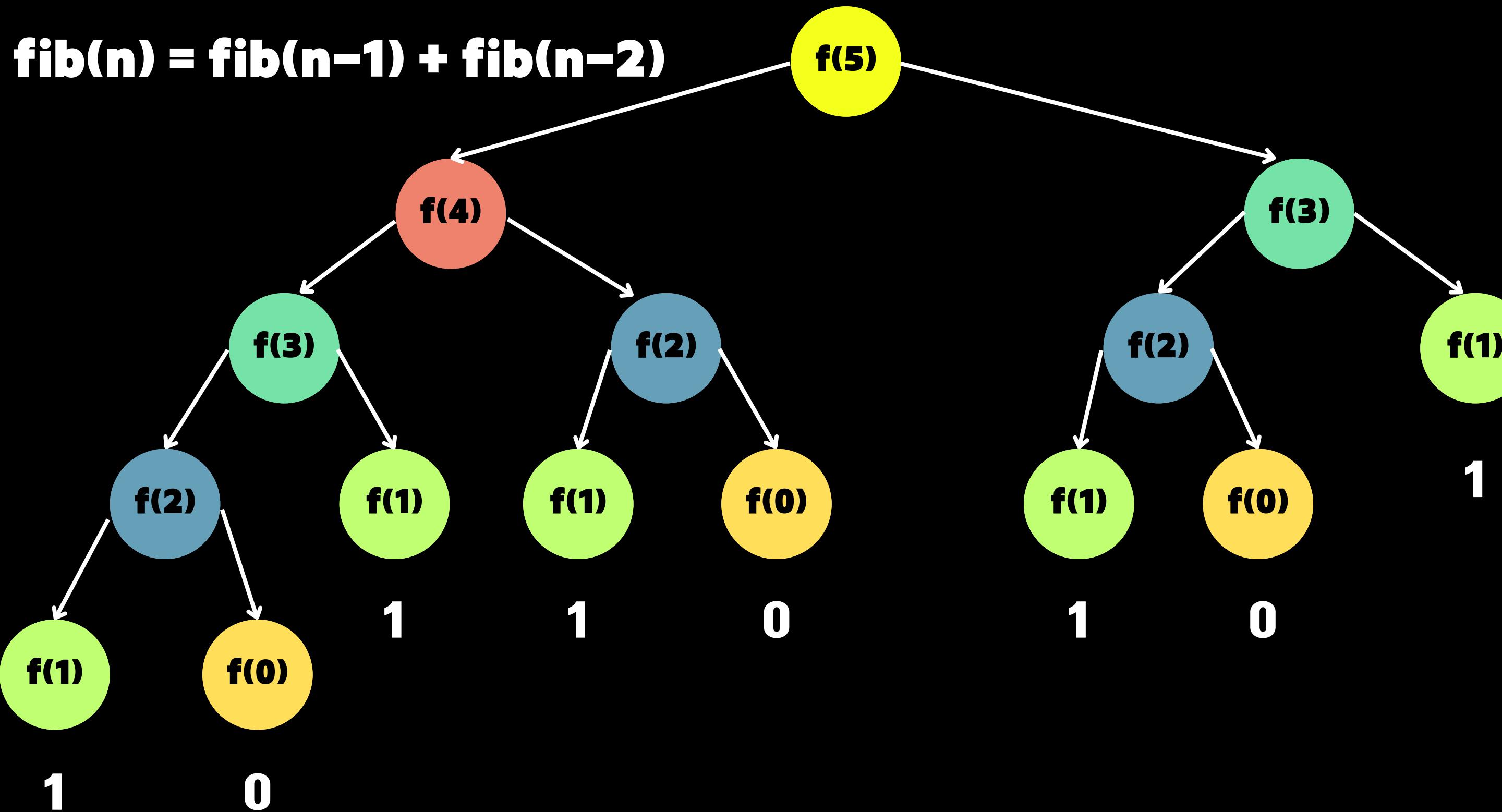


진짜!  
모두를 위한  
자바스크립트



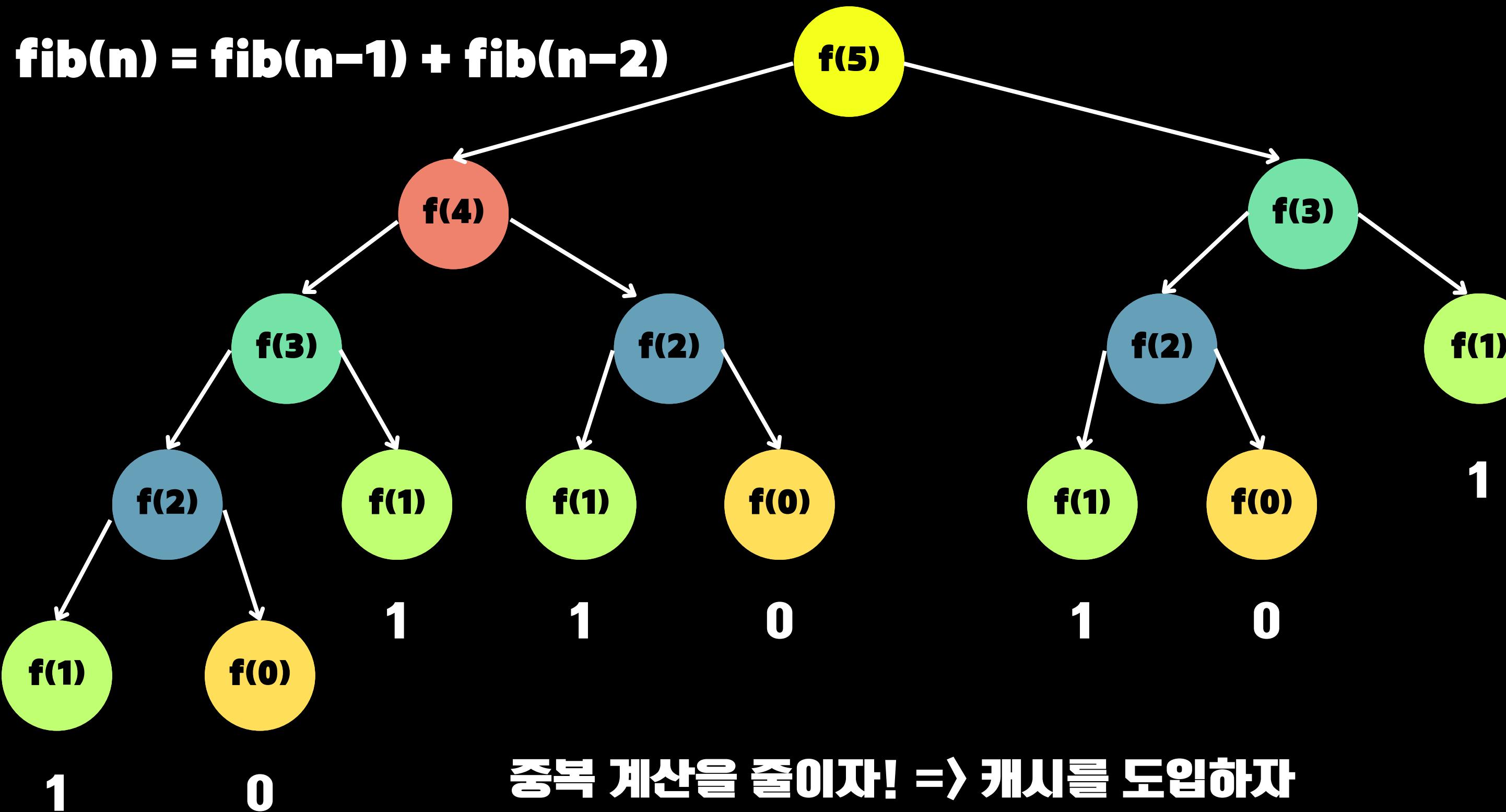
# Execution Context Stack

진짜!  
모두를 위한  
자바스크립트



# Execution Context Stack

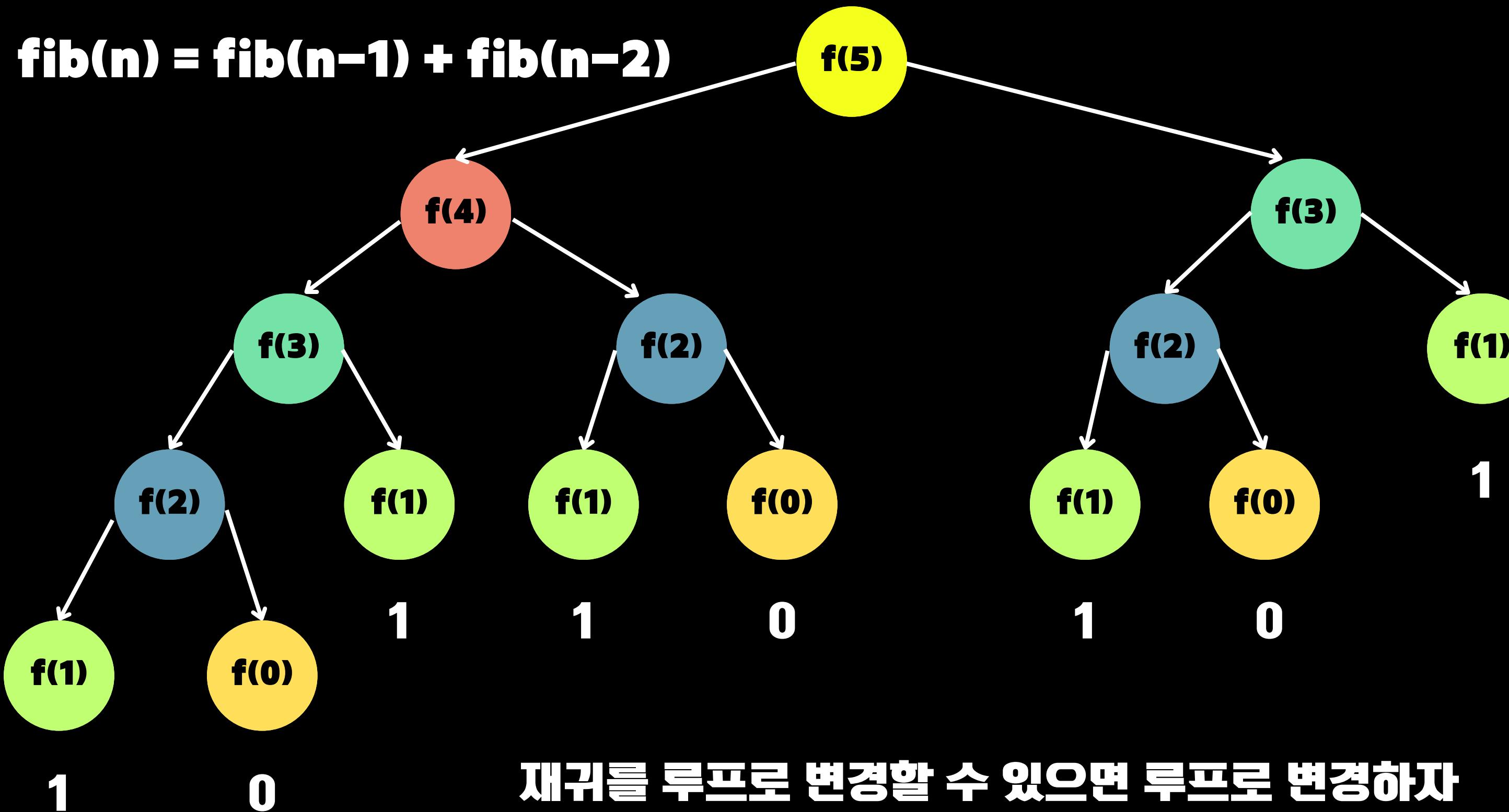
진짜! 모두를 위한  
자바스크립트

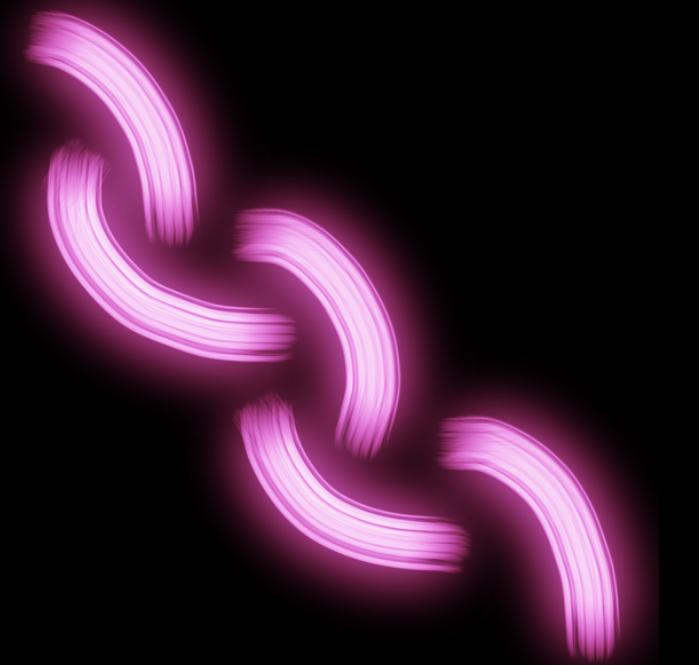


중복 계산을 줄이자! => 캐시를 도입하자  
=> 메모이제이션

# Execution Context Stack

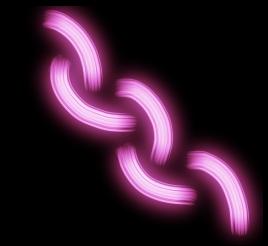
진짜! 모두를 위한  
자바스크립트





# Scope Chain

# Scope Chaining



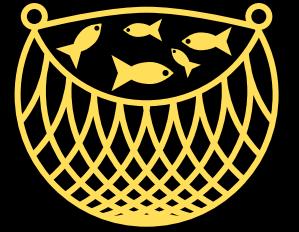
진짜! 모두를 위한  
자바스크립트

- ① **함수 코드에서 참조하는 변수를  
로컬 스코프부터 상위 스코프를 차례대로 방문하여 참조하는 변수를 찾는 것.**
- ② **스코프 체인의 특정 스코프에서 참조 변수를 발견하면 해당 변수의 값을  
사용하고 스코프 체이닝을 통한 참조 변수 검색을 종료한다.**
- ③ **마지막 스코프까지 가서도 참조 변수를 찾지 못하면  
참조 오류(Reference Error)가 발생한다.**

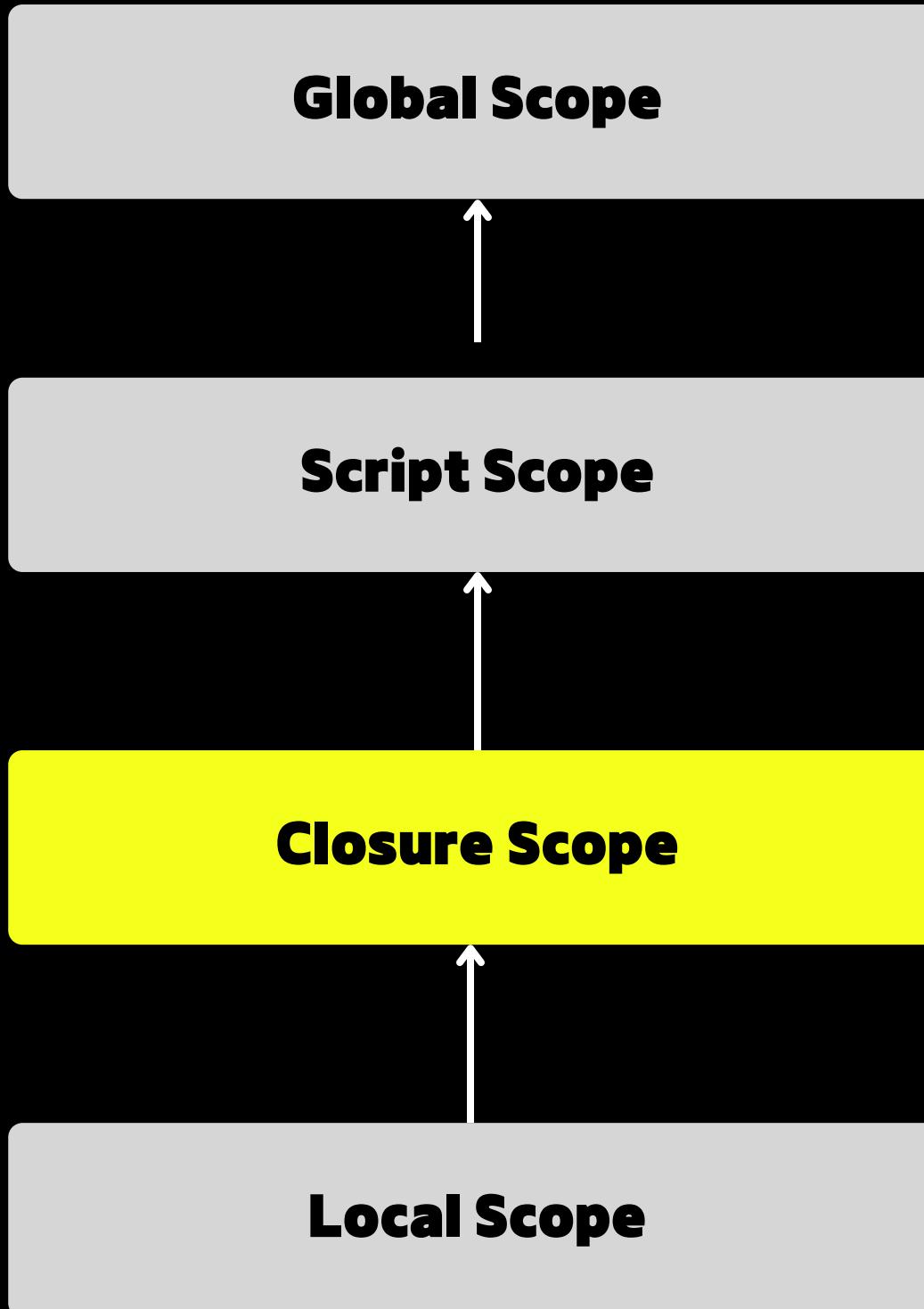


# Closure

# Closure



진짜! 모두를 위한  
자바스크립트

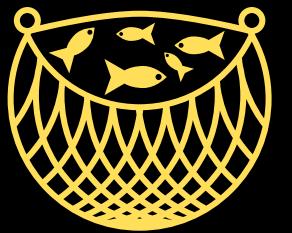


스코프 체인에 클로저 스코프가 있는 함수를  
자바스크립트에서 클로저라고 한다.

그런 어떤 함수가 클로저 스코프를 가질까?



# Closure



진짜! 모두를 위한  
자바스크립트



Closure

```
function outer() {  
    const a = 10;  
    function inner() {  
        console.log({ a });  
    }  
    inner();  
}  
outer();
```

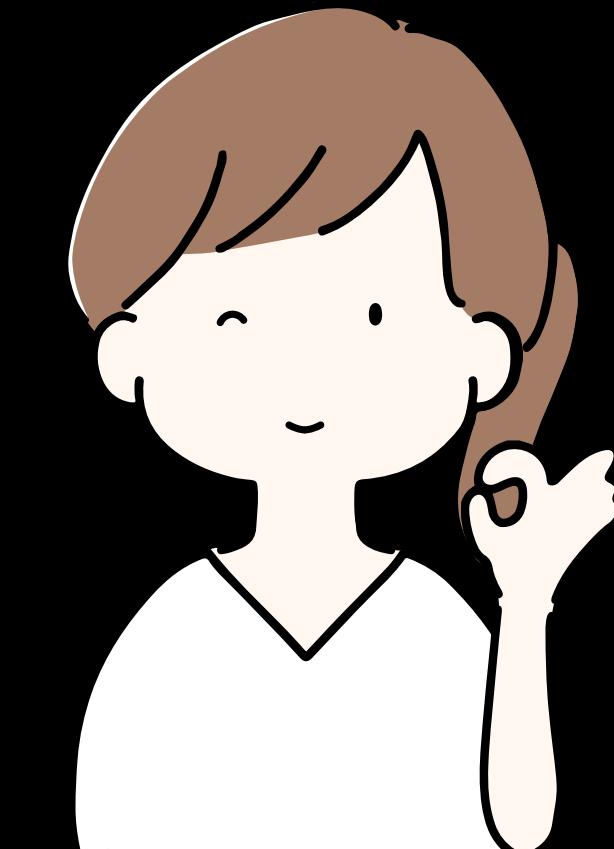
```
1 function outer() {  
2     const a = 10;  
3     function inner() {  
4         console.log({ a });  
5     }  
6     inner();  
7 }  
8 outer();
```

Paused on breakpoint

- ▶ Watch
- ▶ Breakpoints
- ▼ Scope
- ▼ Local
  - ▶ this: Window
- ▼ Closure (outer)
  - a: 10
- ▶ Global
- ▼ Call Stack
- ▶ inner
- outer
- (anonymous)

{ } Line 4, Column 9 Coverage: n/a

어떤 함수의 내부 중첩 함수가 상위 스코프의 변수를  
참조하면 상위 스코프에 대한 클로저 스코프를 생성한다.



# Closure



진짜! 모두를 위한  
자바스크립트



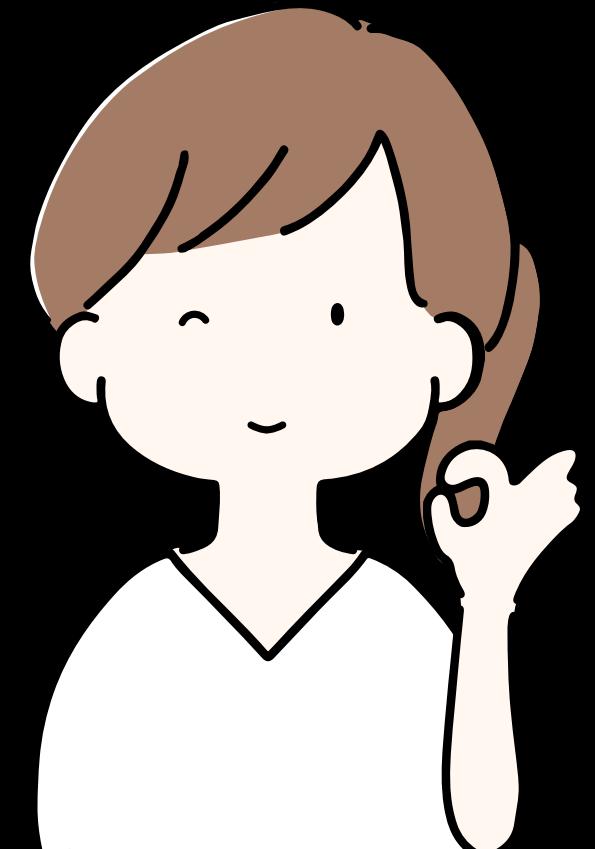
Closure

```
function outer() {  
    const a = 10;  
    function inner() {  
        console.log('Hello');  
    }  
    inner();  
}  
outer();
```

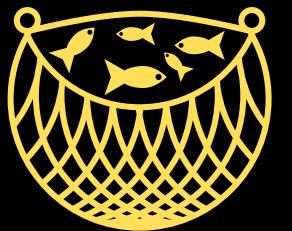
```
1 function outer() {  
2     const a = 10;  
3     function inner() {  
4         console.log('Hello');  
5     }  
6     inner();  
7 }  
8 outer();
```

▶ Paused on breakpoint  
▶ Watch  
▶ Breakpoints  
▼ Scope  
▼ Local  
▶ this: Window  
▶ Global  
▼ Call Stack  
▶ inner  
outer  
(anonymous)

그러나 어떤 함수의 내부 중첩 함수가 상위 스코프의 변수를 참조하지 않으면 클로저 스코프를 생성하지 않는다.



# Closure



진짜!  
모두를 위한  
자바스크립트

• • •

Closure

```
function outer() {  
    const a = 10;  
    const b = 20;  
    function inner() {  
        console.log(a);  
    }  
    inner();  
}  
outer();
```

```
1 function outer() {  
2     const a = 10;  
3     const b = 20;  
4     function inner() {  
5         console.log(a);  
6     }  
7     inner();  
8 }  
9 outer();
```

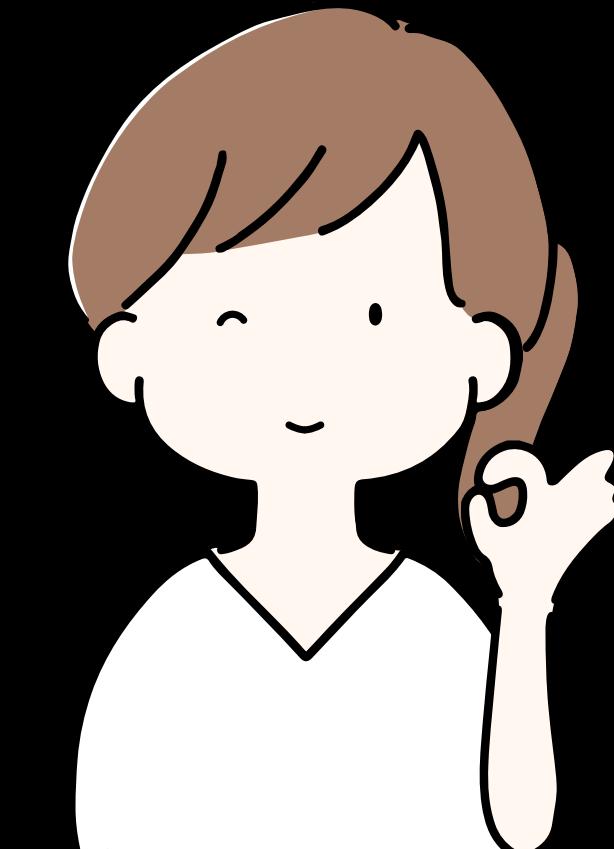
{ } Line 5, Column 9

Coverage: n/a

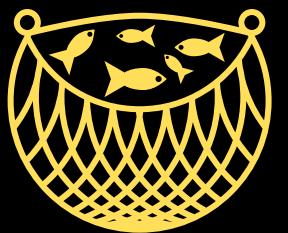
● Paused on breakpoint  
▶ Watch  
▶ Breakpoints  
▼ Scope  
▼ Local  
▶ this: Window  
▼ Closure (outer)  
 a: 10  
▶ Global  
▼ Call Stack  
▶ inner  
outer  
(anonymous)

클로저 스코프는 실제로 참조하는 값만 클로저 스코프에  
기록 또는 캡쳐를 한다.

메모리를 절약하기 위해서다.



# Closure



진짜! 모두를 위한  
자바스크립트

• • •

Closure

```
function outer() {  
    const a = 10;  
    const b = 20;  
    function inner() {  
        console.log(a);  
    }  
    return inner;  
}  
const f = outer();  
f();
```

```
1 function outer() {  
2     const a = 10;  
3     const b = 20;  
4     function inner() {  
5         console.log(a);  
6     }  
7     return inner;  
8 }  
9 const f = outer();  
10 f();
```

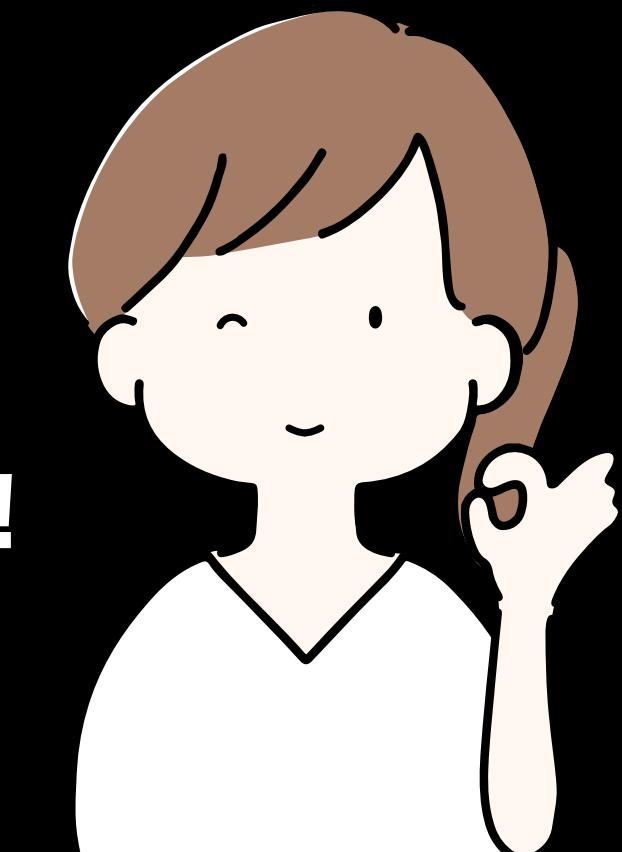
{ } Line 9, Column 1

Coverage: n/a

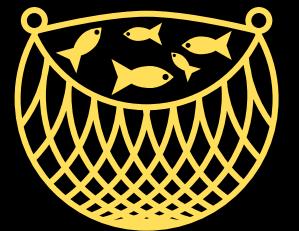
● Paused on breakpoint  
▶ Watch  
▶ Breakpoints  
▼ Scope  
 ▼ Local  
 ▶ this: Window  
 ▼ Closure (outer)  
 a: 10  
 ▼ Script  
 ▶ f: f inner()  
 ▶ Global  
 ▼ Call Stack  
 ▶ inner  
 (anonymous)

클로저의 강력함은 바로 자신이 정의된 스코프를 벗어 날 수  
있다는 점이다.

outer 함수의 실행 컨텍스트가 종료 되었음에도  
inner 함수의 로직을 outer 함수 외부에서 실행할 수 있다!



# Closure



진짜! 모두를 위한  
자바스크립트

```
...  
Closure  
  
function outer() {  
    const a = 10;  
    const b = 20;  
    function inner(n) {  
        console.log(a * n);  
    }  
    return inner;  
}  
const f = outer();  
f(10);
```

```
1 function outer() {  
2     const a = 10;  
3     const b = 20;  
4     function inner(n) { n = 10  
5         console.Dlog(a * n);  
6     }  
7     return inner;  
8 }  
9 const f = outer();  
10 f(10);
```

Paused on breakpoint

- ▶ Watch
- ▶ Breakpoints
- ▼ Scope
- ▼ Local
  - ▶ this: Window
  - n: 10
- ▼ Closure (outer)
  - a: 10
- ▼ Script
  - ▶ f: f inner(n)
- ▶ Global
- ▼ Call Stack

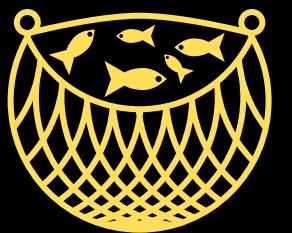
{ } Line 5, Column 9 Coverage: n/a

**outer 의 로컬 스코프에 있던 변수 a 도 자신이 정의된  
스코프를 벗어나 사용될 수 있다!**

**outer 함수의 실행 컨텍스트가 종료 되었음에도  
outer 함수의 로컬 변수 a 는 계속 살아 있다.**



# Closure



진짜! 모두를 위한  
자바스크립트

```
...  
Closure  
  
function outer() {  
    const a = 10;  
    const b = 20;  
    function inner(n) {  
        console.log(a * n);  
    }  
    return inner;  
}  
const f = outer();  
f(10);
```

```
1 function outer() {  
2     const a = 10;  
3     const b = 20;  
4     function inner(n) { n = 10  
5         console.Dlog(a * n);  
6     }  
7     return inner;  
8 }  
9 const f = outer();  
10 f(10);
```

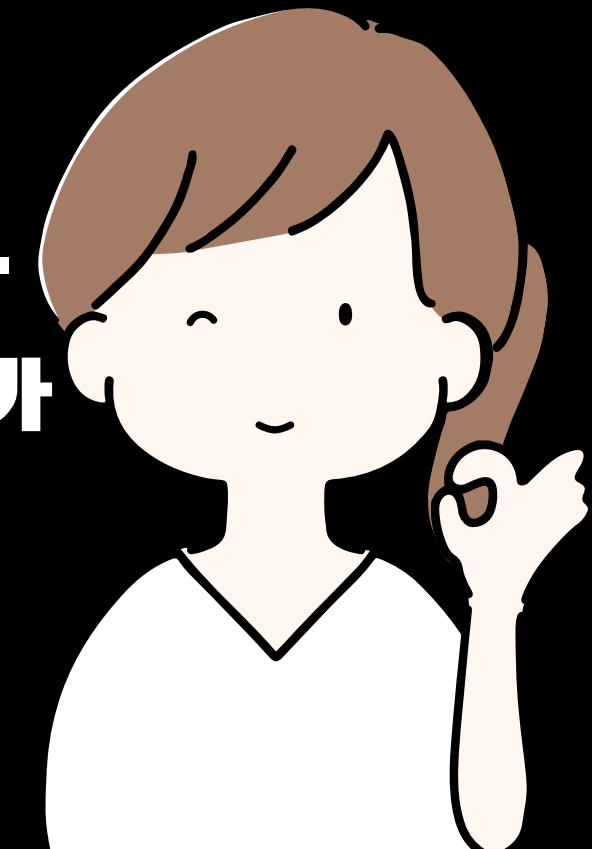
Paused on breakpoint

- ▶ Watch
- ▶ Breakpoints
- ▼ Scope
- ▼ Local
  - ▶ this: Window
  - n: 10
- ▼ Closure (outer)
  - a: 10
- ▼ Script
  - ▶ f: f inner(n)
  - ▶ Global
- ▼ Call Stack

{ } Line 5, Column 9 Coverage: n/a ▶ inner

**outer함수 스코프에 대한 클로저 스코프가 클로저 inner가 참조하는 변수를 모두 strong 참조 또는 캡쳐하기 때문이다.**

**그래서 outer함수가 종료되면 outer함수의 실행 컨텍스트가 메모리에서 정리되고 b 변수가 G.C되지만 a 변수는 클로저 스코프가 strong 참조로 캡쳐하고 있기 때문에 outer 함수 스코프를 벗어나 계속 살아 있을 수 있는 것이다.**



# Closure



진짜! 모두를 위한  
자바스크립트

```
...  
Closure  
  
function outer() {  
    const a = 10;  
    const b = 20;  
    function inner(n) {  
        console.log(a * n);  
    }  
    return inner;  
}  
const f = outer();  
f(10);
```

```
1 function outer() {  
2     const a = 10;  
3     const b = 20;  
4     function inner(n) { n = 10  
5         console.Dlog(a * n);  
6     }  
7     return inner;  
8 }  
9 const f = outer();  
10 f(10);
```

Paused on breakpoint

- ▶ Watch
- ▶ Breakpoints
- ▼ Scope
- ▼ Local
  - ▶ this: Window
  - n: 10
- ▼ Closure (outer)
  - a: 10
- ▼ Script
  - ▶ f: f inner(n)
  - ▶ Global
- ▼ Call Stack

{ } Line 5, Column 9 Coverage: n/a ▶ inner

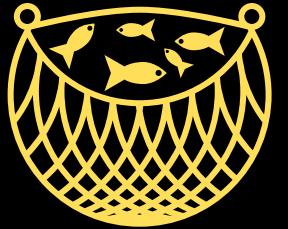
이러한 변수를 자유변수라고 부른다.

원래 선언된 위치였던 상위 함수의 스코프를 벗어나 자유로  
이 살고 있기 때문에 "자유롭다" 라고 할 수 있다.

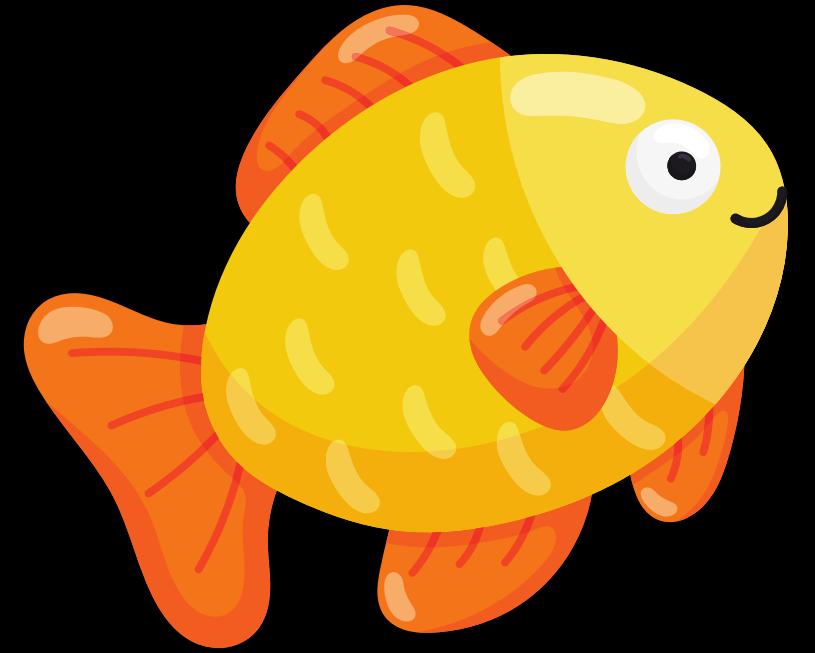
원래 자신이 선언된 함수를 벗어나 다른 외부 공간(스코프, 함수)  
에서도 "자유롭게" 접근 가능하고 변경이 가능하다.



# Closure



진짜!  
모두를 위한  
자바스크립트



자유변수와 클로저를 사용해 다양한 방법으로 활용할 수 있다!

