



자바스크립트 심화

01 실행 컨텍스트



커리큘럼



실행 컨텍스트

실행 컨텍스트는 자바스크립트의 여러 개념을 깊게 이해하기 위해 중요한 개념입니다. 자바스크립트 스펙에도 명시되어 있어, 실행 컨텍스트를 이해하면 자바스크립트의 동작을 스펙 레벨에서 이해하고 활용할 수 있습니다.



자바스크립트 실행

자바스크립트는 흔히 인터프리터 언어라고 말하지만 컴파일 언어의 면모가 있습니다. 자바스크립트 컴파일은 여러 단계로 구성되며, 각 단계별로 코드를 처리하는 목적이 다릅니다. 자바스크립트의 컴파일 과정을 이해하면 코드가 선언되고 실행될 때 생기는 버그, 메모리 릭(Memory Leak)을 이해하고 고칠 수 있습니다.

커리큘럼



자바스크립트 비동기

자바스크립트는 다른 멀티스레드 프로그래밍 언어와 다르게 싱글 스레드 환경에서 비동기 동작을 처리합니다. 싱글 스레드 비동기 환경을 구성하는 중요 요소 중 하나인 이벤트 루프에 대해 이해하면, 자바스크립트 엔진 레벨에서의 비동기 코드 동작을 이해할 수 있습니다. 복잡한 비동기 동작을 이해하고 높은 수준의 비동기 코드를 작성하며, 버그의 원인을 파악할 수 있게 됩니다.



미니 프로젝트

지금까지 배운 내용을 활용하는 프로젝트를 진행합니다.

추천대상

1. HTML, CSS, JS의 기본 문법과 내용을 알고 있는 분

기초적인 HTML, CSS, JS의 내용을 이해하고 있는 분

2. HTML, CSS, JS를 활용한 프로젝트를 경험해보신 분

프론트엔드 프로젝트를 세팅하고 구현해본 경험이 있는 분

3. 자바스크립트의 필수 문법에 대해 자세히 알고 싶은 분

기초 강의에서 배우지 않은 부분에 대하여 더 자세히 알고 싶은 분

수강목표

1. 자바스크립트의 필수적인 문법을 이해할 수 있다.

기초적인 자바스크립트 문법에 더해, 깊은 이해를 요구하는 자바스크립트의 문법을 이해한다.

2. HTML, CSS, JS로 프로젝트를 구현할 수 있다.

간단한 프론트엔드 프로젝트를 구성하고 구현한다.

3. 자바스크립트의 내부 동작에 대해 이해할 수 있다.

비동기 처리, 변수 생명주기 등의 내부 동작에 대해 이해한다.

목차

- 01. 자바스크립트 함수가 실행되는 과정
- 02. 실행 컨텍스트
- 03. this가 가리키는 것
- 04. 화살표 함수와 일반 함수
- 05. 자바스크립트 Closure
- 06. ES6 Rest, Spread operator

01

자바스크립트 함수가 실행되는 과정



✓ 자바스크립트 코드의 실행 1

code

```
// 어떤 코드도 없는 경우.
```


✓ 자바스크립트 코드의 실행 1

code

```
// 어떤 코드도 없는 경우.
```

this

변수들(Variable Object)

Scope chain

✓ 자바스크립트 코드의 실행 1

code

```
// 어떤 코드도 없는 경우.
```

this : window

변수들(Variable Object) : {}

Scope chain : []

✓ 자바스크립트 코드의 실행 1

- 자바스크립트 엔진은 코드가 없어도 실행 환경(실행 컨텍스트)을 초기화한다.
- 스코프(scope)는 코드가 현재 실행되는 환경, 맥락(context)을 의미한다.
- this 포인터, 스코프에 저장된 변수들, 스코프 체인 등이 환경에 포함된다.
- this 포인터의 경우, 글로벌 스코프에서는 window를 가리킨다.

✓ 자바스크립트 코드의 실행 2

code

```
function myFunc() {  
  let a = 10  
  let b = 20  
  function add(first, second) {  
    return first + second  
  }  
  return add(a, b)  
}
```

myFunc()

✓ 자바스크립트 코드의 실행 2

code

```
function myFunc() {  
  let a = 10  
  let b = 20  
  function add(first, second)  
  {  
    return first + second  
  }  
  return add(a, b)  
}
```

myFunc()

this

변수들(Variable Object)

Scope chain

✓ 자바스크립트 코드의 실행 2

code

```
function myFunc() {  
  let a = 10  
  let b = 20  
  function add(first, second)  
  {  
    return first + second  
  }  
  return add(a, b)  
}
```

myFunc()

this : undefined (strict mode)

변수들(Variable Object) : {
 a : 10
 b : 20
 add : function { ... }
}

Scope chain : [global]

this : window

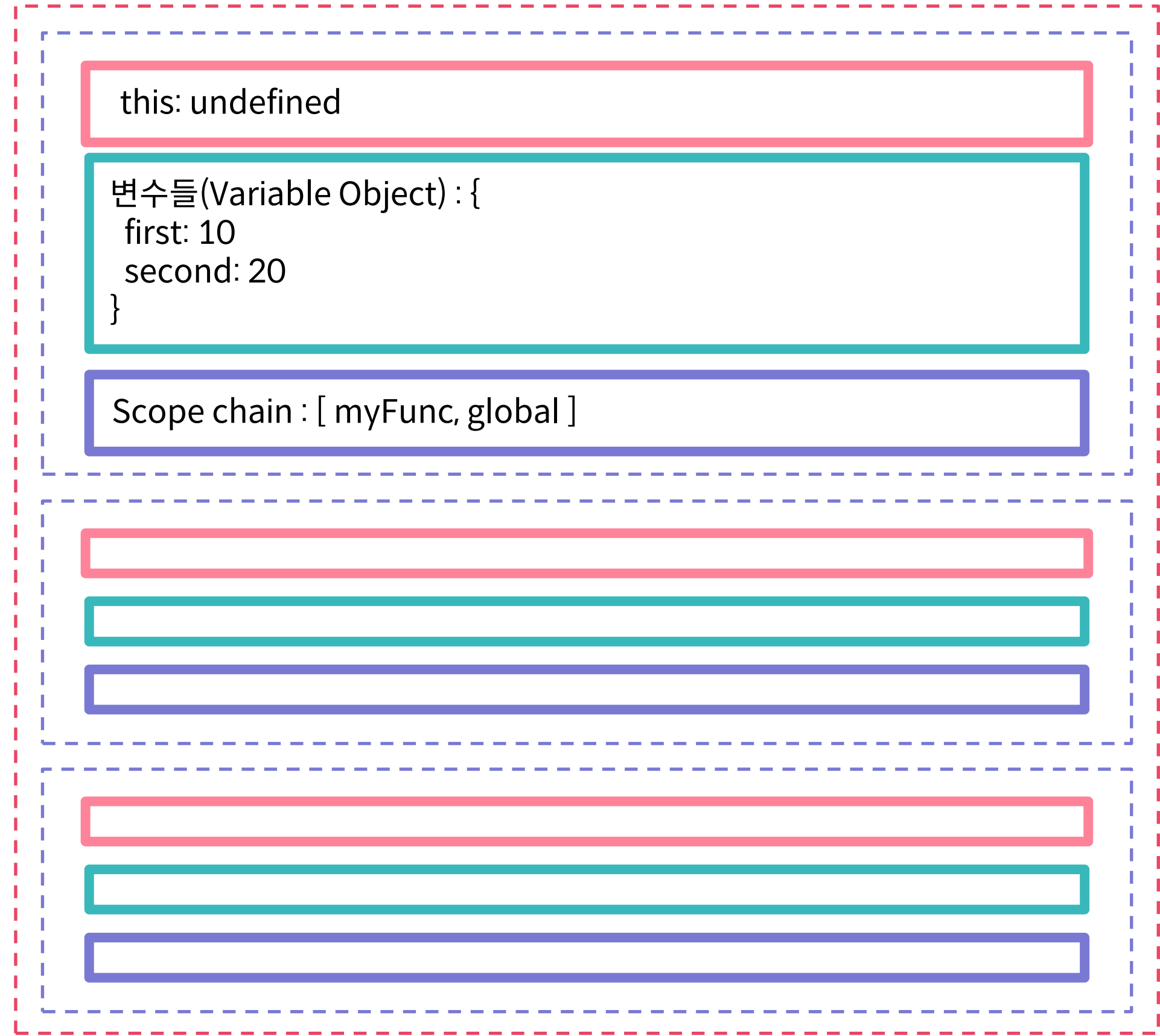
변수들(Variable Object) : {}

Scope chain : []

✓ 자바스크립트 코드의 실행 2

code

```
function myFunc() {  
  let a = 10  
  let b = 20  
  function add(first, second)  
  {  
    return first + second  
  }  
  return add(a, b)  
}  
  
myFunc()
```



✓ 자바스크립트 코드의 실행 2

- 함수가 실행되면, 함수 스코프에 따라 환경이 만들어진다.
- this, 함수 스코프의 변수들, 그리고 스코프 체인이 형성된다.
- 스코프 체인을 따라 글로벌 환경에 도달한다.

✓ 자바스크립트 코드의 실행 3

code

```
let o = {  
  name: 'Daniel',  
  method: function(number) {  
    return this.name.repeat(number)  
  }  
}
```

```
function myFunc() {  
  let n = 10  
  return o.method(n)  
}
```

```
myFunc()
```

✓ 자바스크립트 코드의 실행 3

code

```
let o = {  
  name: 'Daniel',  
  method: function(number) {  
    return this.name.repeat(number)  
  }  
}
```

```
function myFunc() {  
  let n = 10  
  return o.method(n)  
}
```

myFunc()

this: window

변수들(Variable Object) : {
 o : { ... },
 myFunc : function() {...}
}

Scope chain: []

✓ 자바스크립트 코드의 실행 3

code

```
let o = {  
  name: 'Daniel',  
  method: function(number) {  
    return this.name.repeat(number)  
  }  
}
```

```
function myFunc() {  
  let n = 10  
  return o.method(n)  
}
```

```
myFunc()
```

this: undefined

변수들(Variable Object) : {
 n: 10
}

Scope chain : [global]

this: window

변수들(Variable Object) : {
 o: {...},
 myFunc: function() {...}
}

Scope chain : []

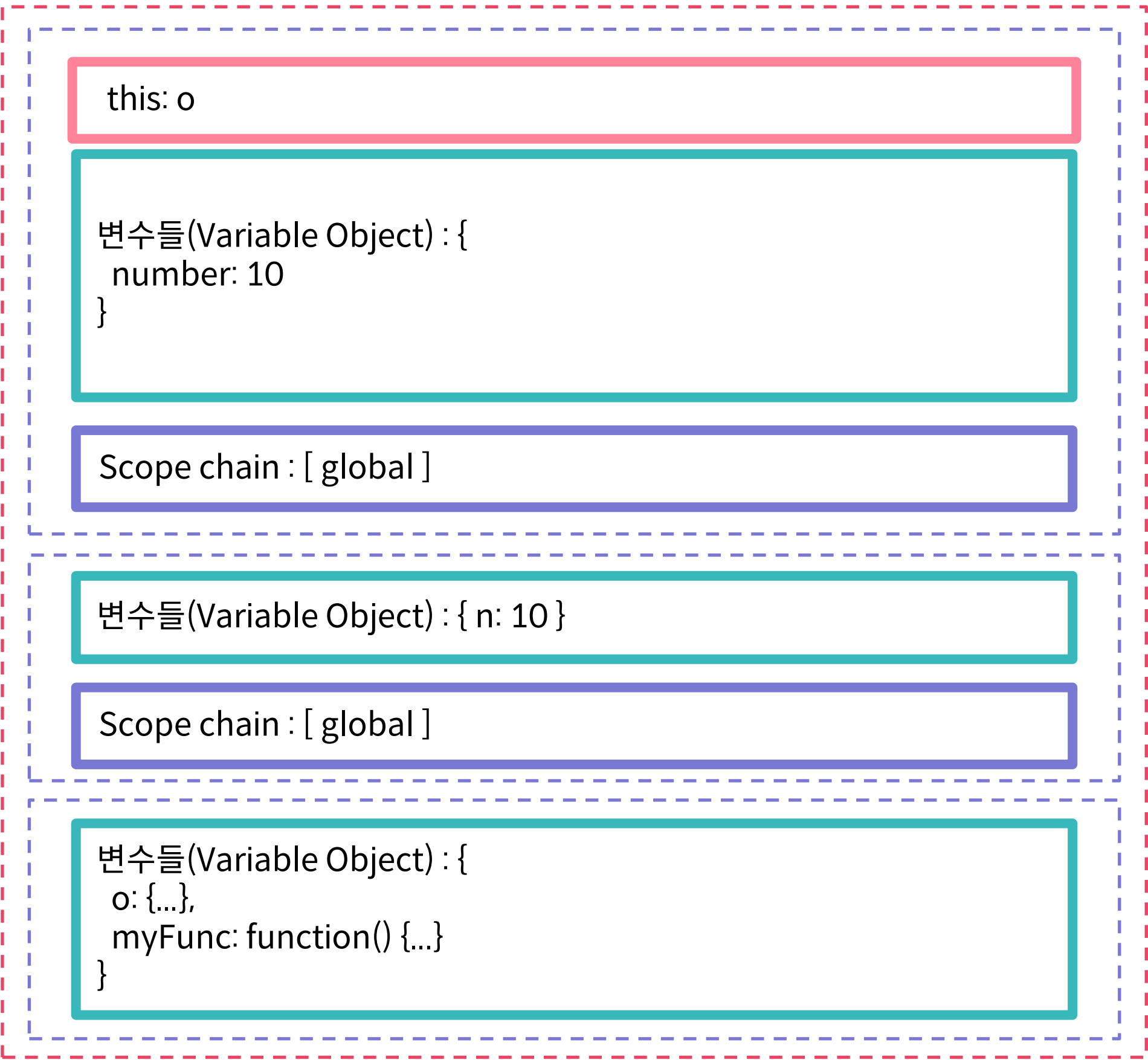
✓ 자바스크립트 코드의 실행 3

code

```
let o = {
  name: 'Daniel',
  method: function(number) {
    return this.name.repeat(number)
  }
}

function myFunc() {
  let n = 10
  return o.method(n)
}

myFunc()
```

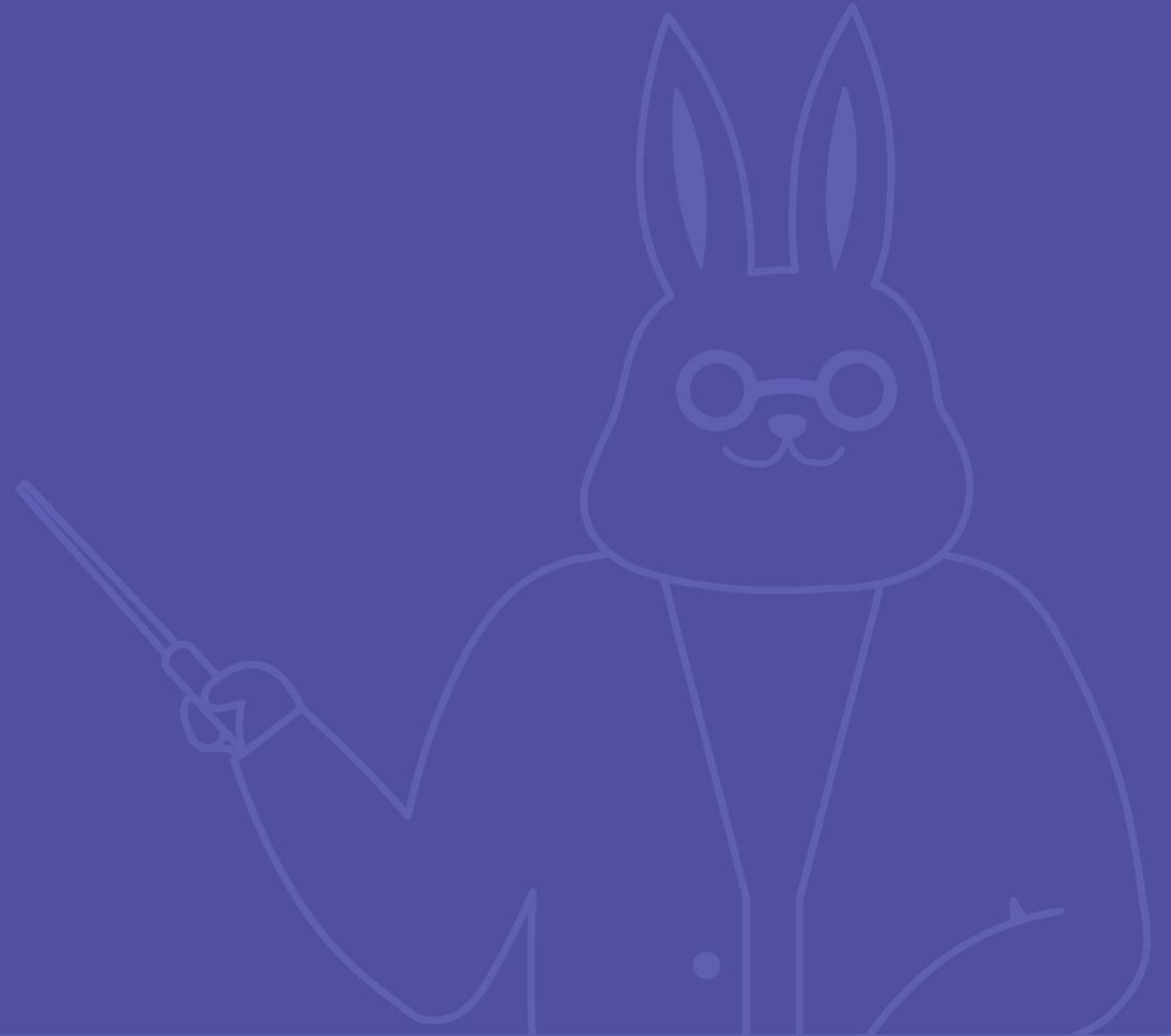


✓ 자바스크립트 코드의 실행 3

- 객체의 메서드의 경우, 메서드 환경의 `this`는 해당 객체를 가리키게 된다.
- 하지만 `this`가 가리키는 것은 환경에 따라 변할 수 있다.

02

실행 컨텍스트



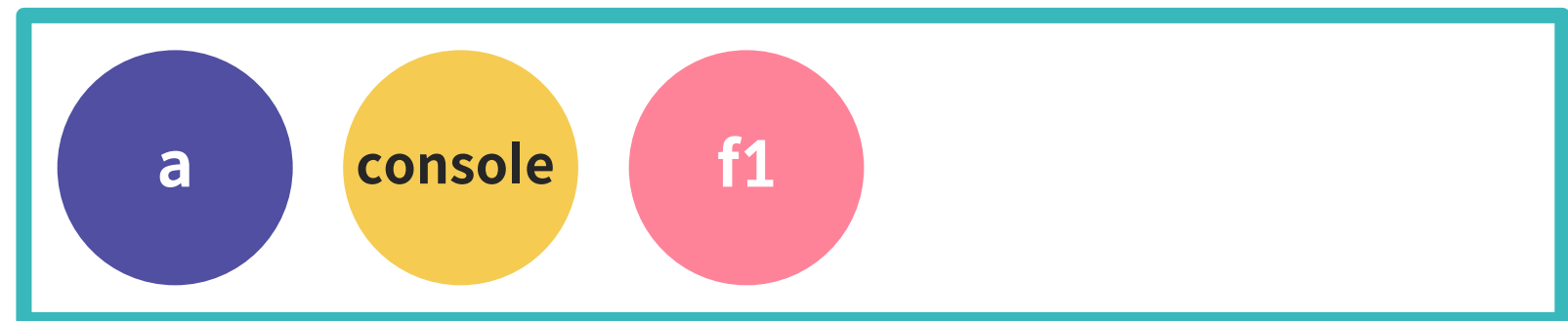
✓ 실행 컨텍스트(Execution context)

- 실행 컨텍스트 혹은 실행 맥락은, 자바스크립트 코드가 실행되는 환경.
- 코드에서 참조하는 변수, 객체(함수 포함), this 등에 대한 레퍼런스가 있다.
- 실행 컨텍스트는 전역에서 시작해, 함수가 호출될 때 스택에 쌓이게 된다.

✓ 실행 컨텍스트 스택

code

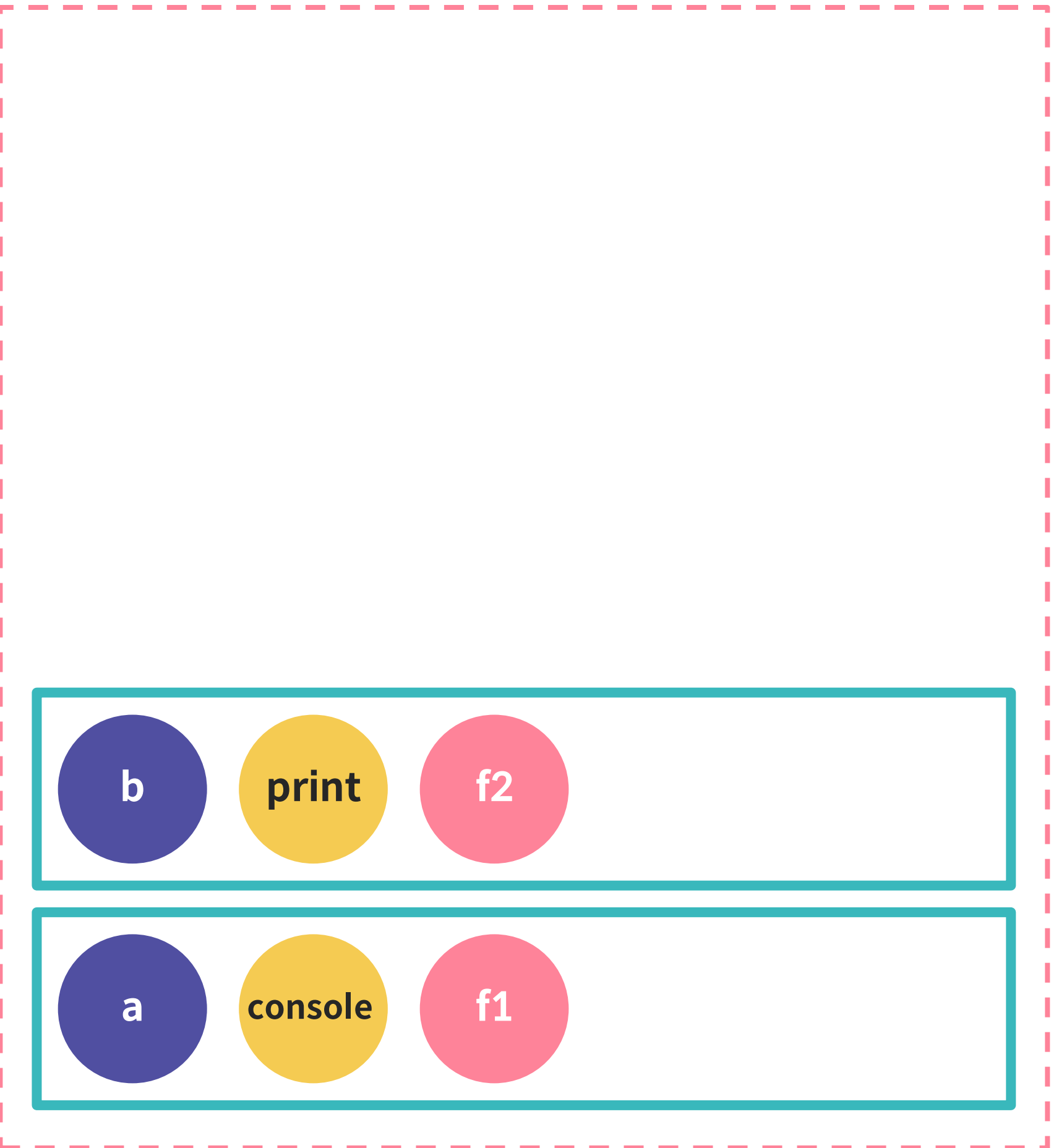
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

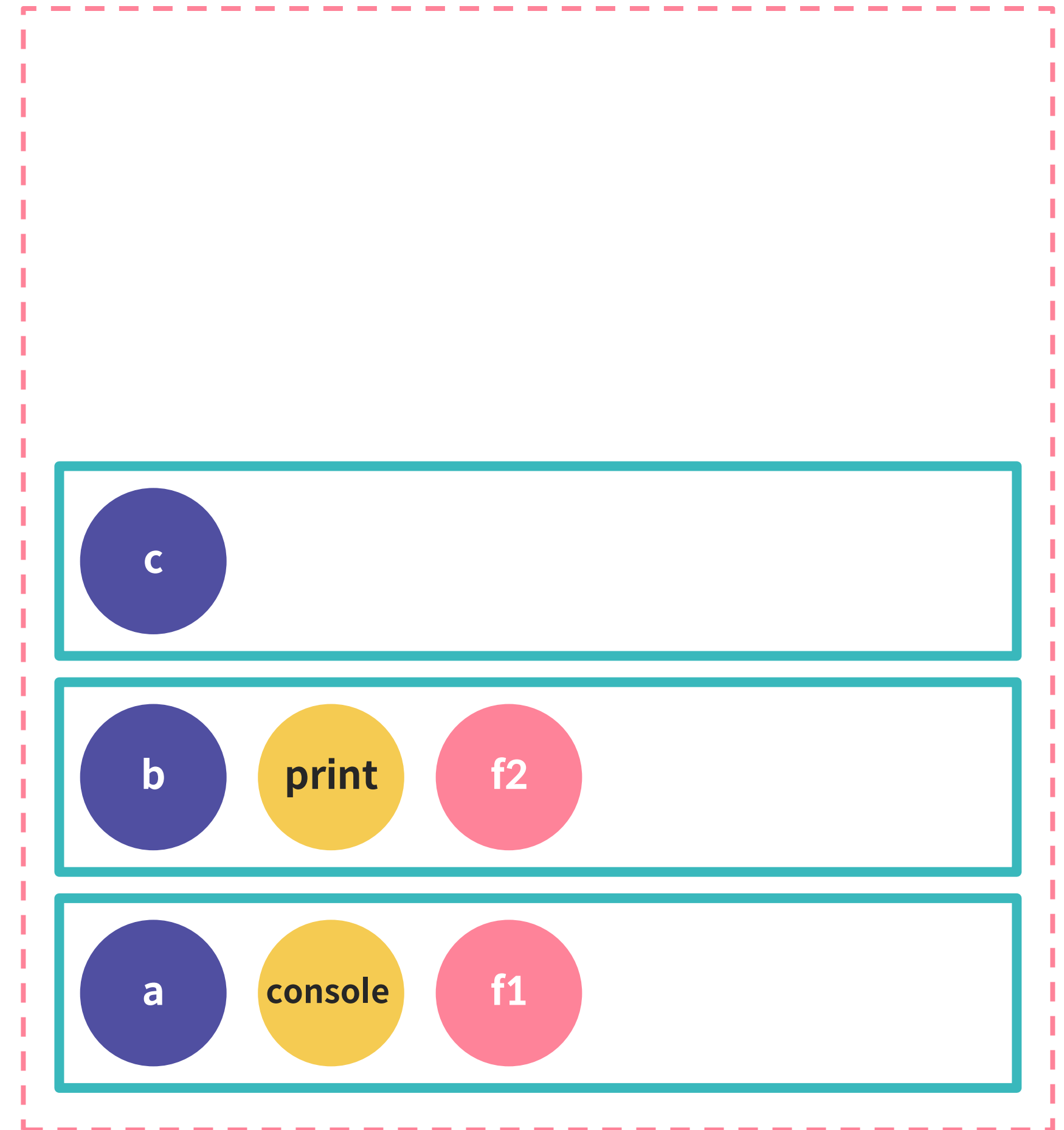
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

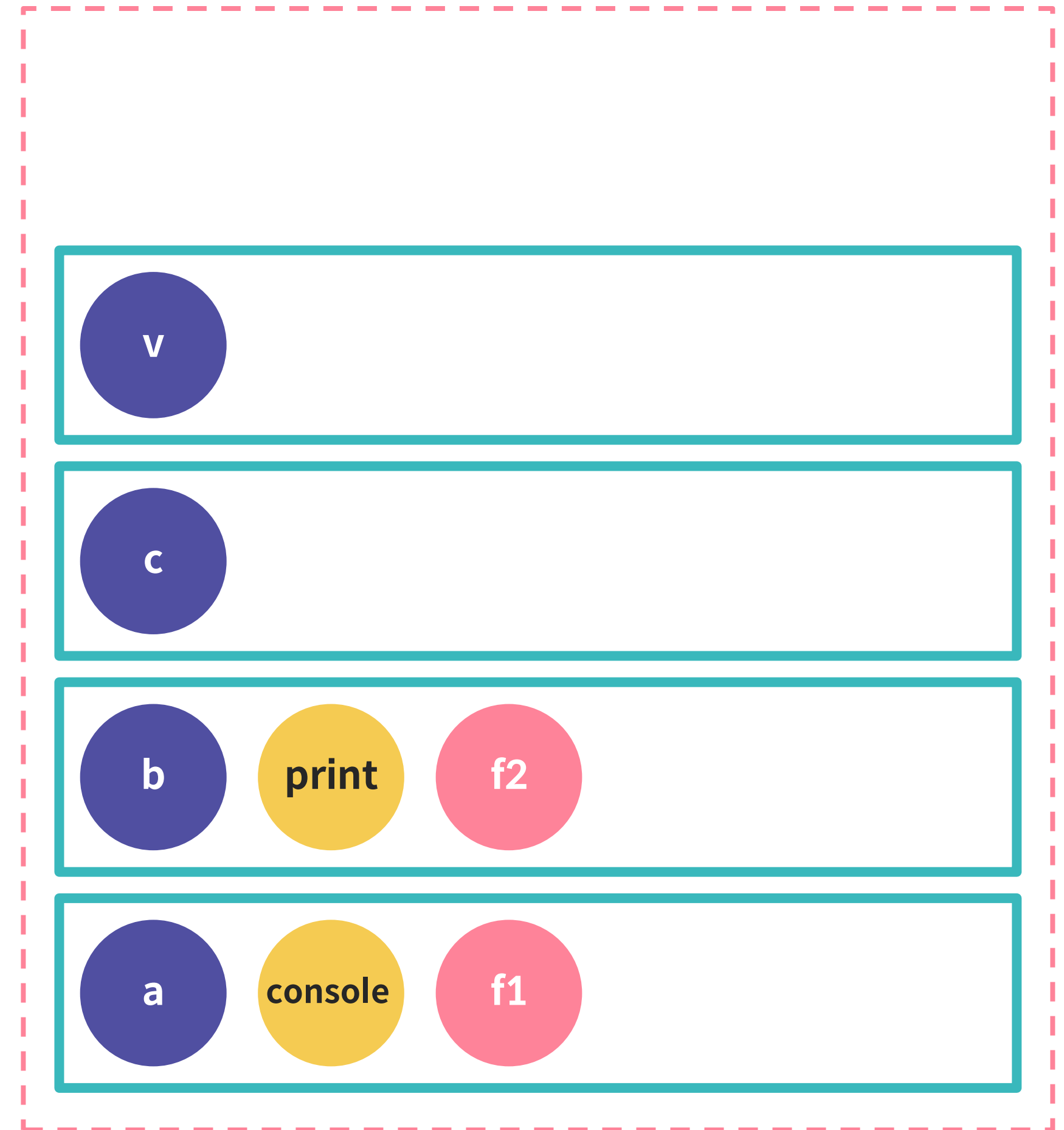
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

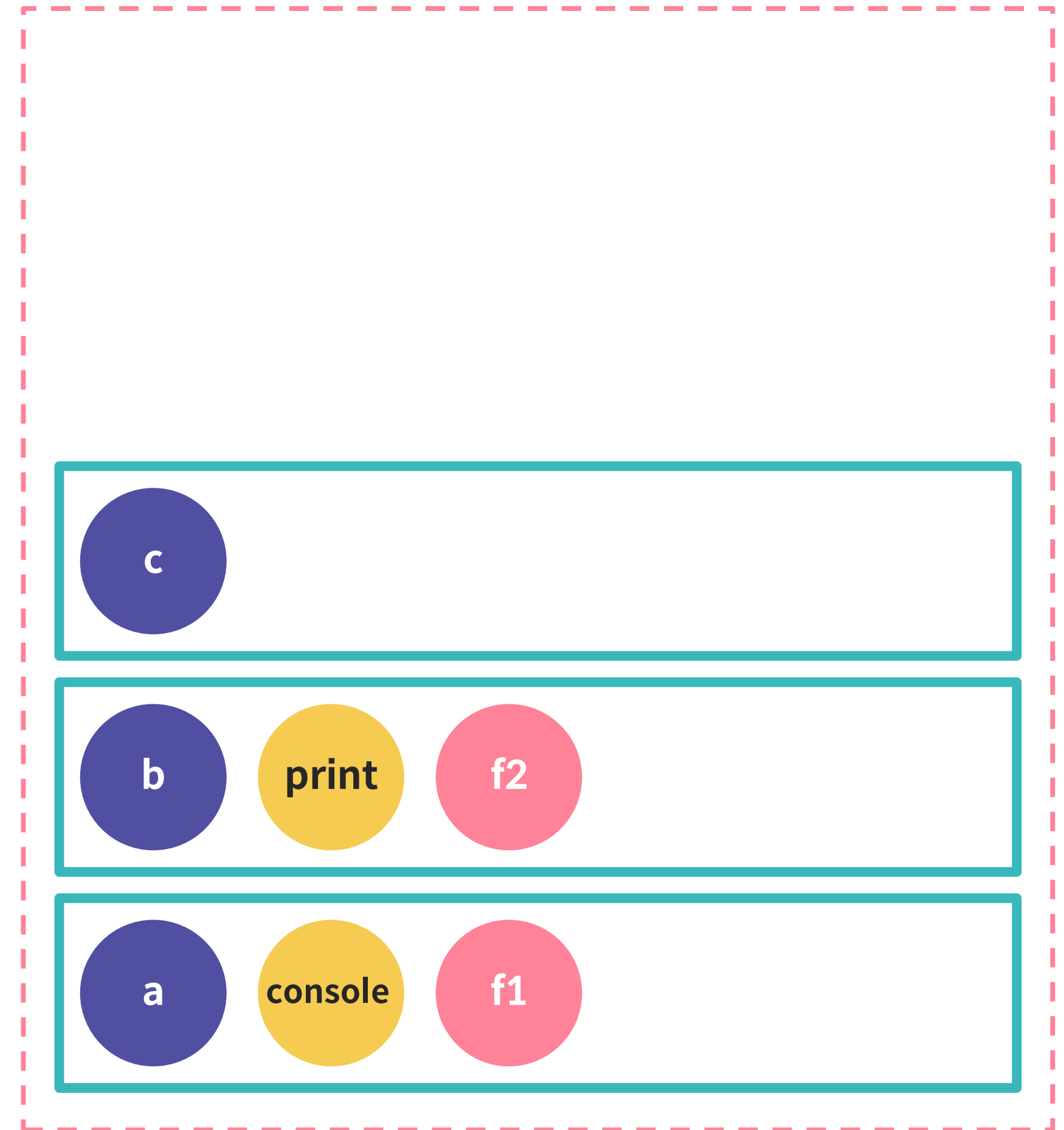
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

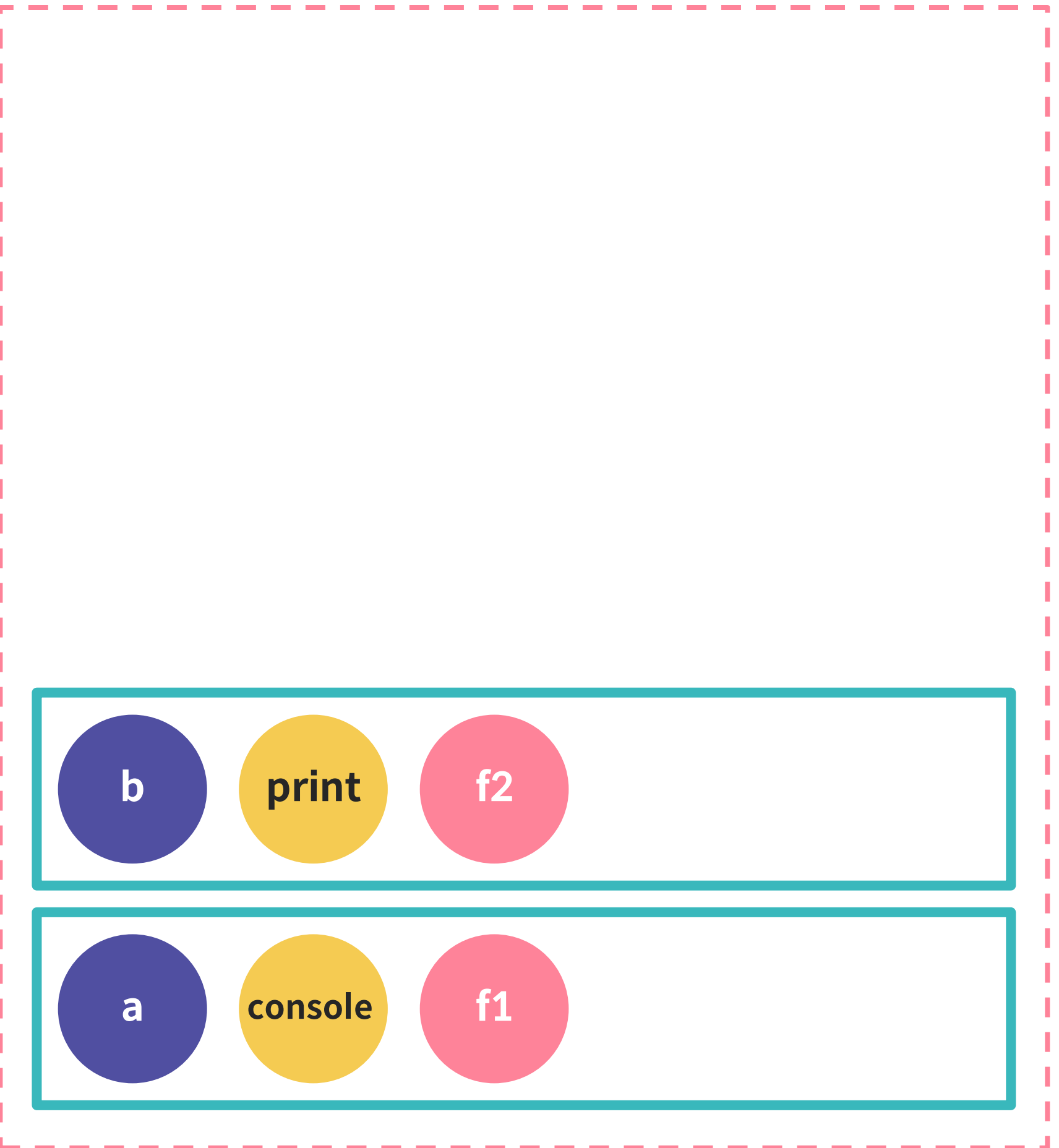
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

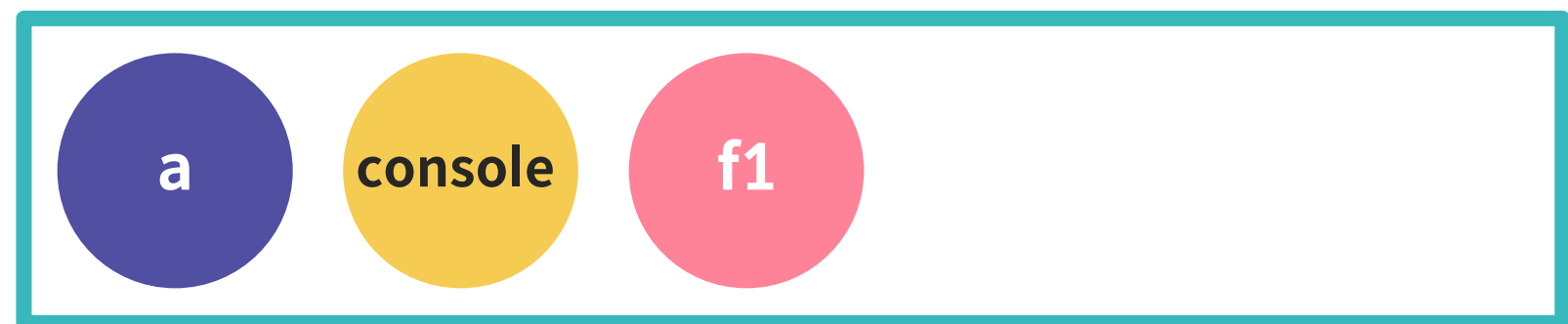
```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 실행 컨텍스트 스택

code

```
let a = 10
function f1() {
  let b = 20
  function print(v) { console.log(v) }
  function f2() {
    let c = 30
    print(a + b + c)
  }
  f2()
}
f1()
```



✓ 전역 실행 컨텍스트, 함수 실행 컨텍스트

- 자바스크립트가 실행될 때 전역 실행 컨텍스트(Global Execution Context)가 만들어진다.
- 함수가 실행될 때 함수 실행 컨텍스트(Function Execution Context)가 만들어진다.

✓ 빈 코드의 실행 컨텍스트

code

```
// 어떤 코드도 없는 경우.
```

전역 실행 컨텍스트

this

변수들(Variable Object)

Scope chain

✓ 함수의 실행 컨텍스트

code

```
function myFunc() {  
  let a = 10  
  let b = 20  
  function add(first, second)  
  {  
    return first + second  
  }  
  return add(a, b)  
}
```

myFunc()

함수 실행 컨텍스트

this : undefined (strict mode)

변수들(Variable Object) : {
 a : 10
 b : 20
 add : function { ... }
}

Scope chain : [global]

전역 실행 컨텍스트

this : window

변수들(Variable Object) : {}

Scope chain : []

03

this가 가리키는 것



✓ dynamic binding

- 함수가 호출되는 상황은 4가지가 있다.
- 함수 호출 - 함수를 직접 호출한다.
- 메서드 호출 - 객체의 메서드를 호출한다.
- 생성자 호출 - 생성자 함수를 호출한다.
- 간접 호출 - call, apply 등으로 함수를 간접 호출한다.

✓ 함수의 호출

- 그 외 콜백 함수의 호출이 있다.
- 콜백 함수는 특정 동작 이후 불려지는 함수이다.
- 보통 다른 함수의 인자로 보내지는 함수를 의미한다.

✓ dynamic binding

code

```
function myFunc() {  
  console.log('myFunc called')  
}
```

myFunc() // 함수를 직접 호출.

```
const o = {  
  name : 'Daniel',  
  printName : function() {  
    console.log(this.name) }  
}
```

o.printName() // 객체의 메서드를 호출.

```
function Person(name) {  
  this.name = name  
  this.printName = function() {  
    console.log(this.name) }  
}
```

const p = new Person('Daniel') // 생성자 호출

```
setTimeout(p.printName.bind(p), 1000)  
// 간접 호출
```

✓ dynamic binding

- 함수는 이렇듯 다양한 상황(환경)에서 호출될 수 있다.
- 함수의 호출 환경에 따라 this는 동적으로 세팅된다.
- 이렇게 this가 환경에 따라 바뀌는 것을 동적 바인딩(dynamic binding)이라 한다.
- bind, apply, call 등으로 this가 가리키는 것을 조작할 수 있다.

✓ dynamic binding

code

```
let o = {
  name: "Daniel",
  f1: () => {
    console.log("[f1] this : ", this);
  },

  f2: function () {
    console.log("[f2] this : ", this);
  },
};

o.f1(); // global
o.f2(); // o

setTimeout(o.f1, 10); // global
setTimeout(o.f2, 20); // global
```

- f1은 화살표 함수로 호출 시 this는 함수가 생성된 환경을 가리키도록 고정된다.
- f2는 일반 함수로 this는 함수를 호출된 환경을 가리키며 this는 동적으로 바뀔 수 있다.
- f2는 객체의 메서드로 호출될 때 객체가 this로 할당된다.

✓ dynamic binding

code

```
let o = {
  name: "Daniel",
  f1: () => {
    console.log("[f1] this : ", this);
  },

  f2: function () {
    console.log("[f2] this : ", this);
  },
};

o.f1(); // global
o.f2(); // o

setTimeout(o.f1, 10); // global
setTimeout(o.f2, 20); // global
```

- 최상단 스코프의 실행 컨텍스트는 전역이다.
- setTimeout으로 함수의 실행 환경을 바꾼다.

✔ this를 조작하는 경우

code

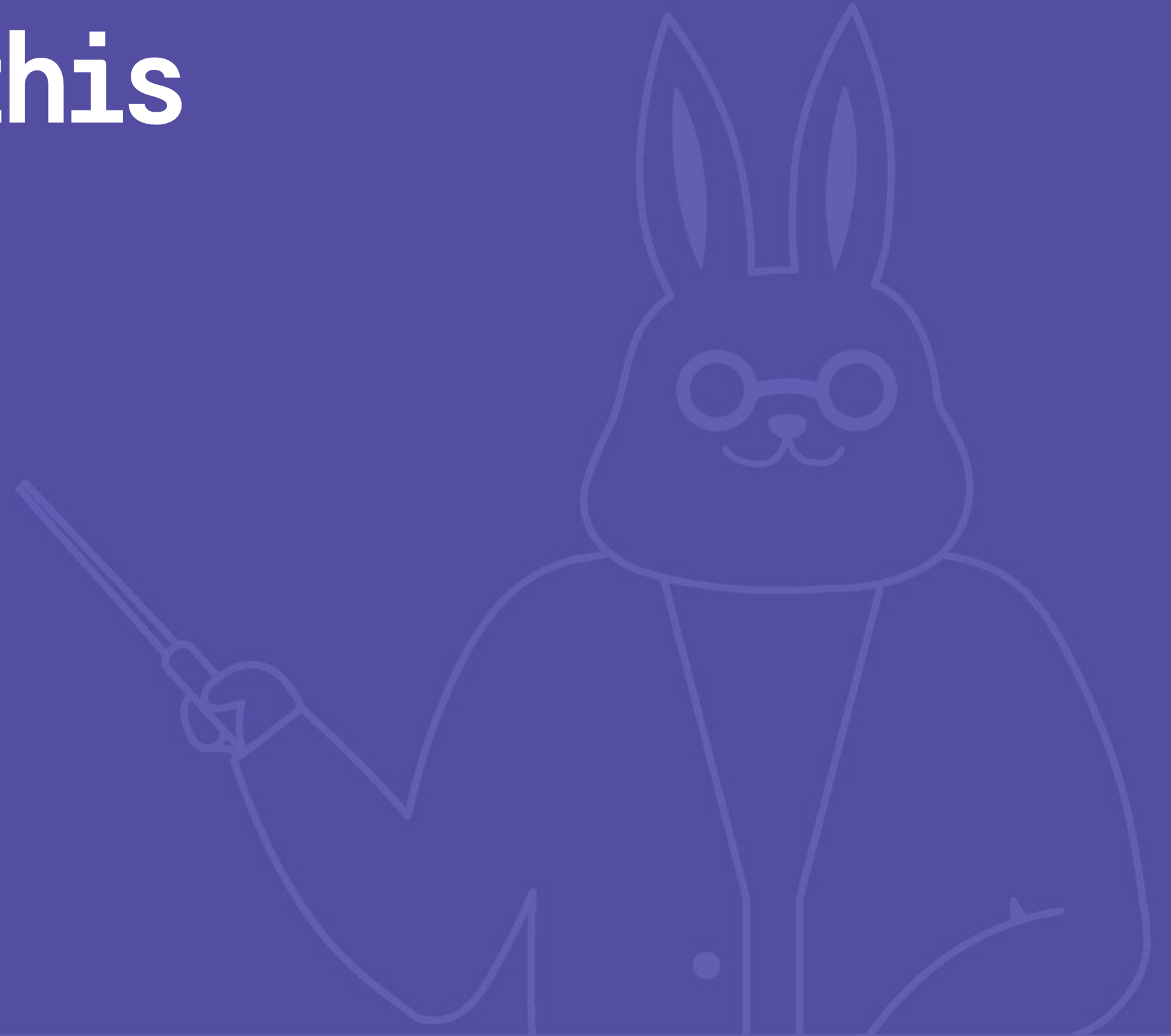
```
let o = {
  name: "Daniel",
  printName: function () {
    console.log("내 이름은 ", this.name);
  },
};

o.printName(); // 내 이름은 Daniel
setTimeout(o.printName, 10); // 내 이름은
undefined
setTimeout(o.printName.bind(o), 20); // 내
이름은 Daniel
```

- bind, call, apply 등의 함수로 this를 조작한다.
- setTimeout은 함수 호출과는 다른 콜백 호출이다.
- printName 메서드는 bind 함수를 이용해 this 변수가 o를 가리키도록 컨텍스트를 동적 바인딩한다.

04

화살표 함수와 일반 함수의 this



✓ 화살표 함수 vs 일반 함수

- 화살표 함수의 this는 호출된 함수를 둘러싼 실행 컨텍스트를 가리킨다.
- 일반 함수의 this는 새롭게 생성된 실행 컨텍스트를 가리킨다.

✓ 화살표 함수 vs 일반 함수

code

```
const o = {
  method() {
    console.log("context : ", this) // o
    let f1 = function () {
      console.log("[f1] this : ", this)
    }
    let f2 = () =>
      console.log("[f2] this : ", this)
    f1() // global
    f2() // o
  },
};
o.method()
```

- f1()은 실행될 때 새로운 컨텍스트를 생성한다.
- 이때 f1에 바인딩된 컨텍스트가 없으므로 this는 global을 가리킨다.
- f2()는 함수 컨텍스트를 생성하며 this 변수는 부모의 컨텍스트를 가리킨다.
- 따라서 this는 o가 된다.

✓ 화살표 함수와 dynamic binding

- 화살표 함수의 this는 정해지면 바꿀 수 없다.
- call, bind, apply를 사용해도 바뀌지 않는다.
- setTimeout 등 this가 바뀌는 상황에서 유용하다.

✓ 화살표 함수와 dynamic binding

code

```
window.name = 'Daniel'
let o = { name : 'Kim' }

let arrowFunction = (prefix) => console.log(prefix + this.name)

arrowFunction('Dr. ') // Dr. Daniel
arrowFunction.bind(o)('Dr. ') // Dr. Daniel
arrowFunction.call(o, 'Dr. ') // Dr. Daniel
arrowFunction.apply(o, ['Dr. ']) // Dr. Daniel
```

05

자바스크립트 Closure



✓ 함수는 일급 객체(first-class object)

- 일급 객체란, 다른 변수처럼 대상을 다룰 수 있는 것을 말한다.
- 자바스크립트에서 함수는 일급 객체이다.
- 즉, 자바스크립트에서 함수는 변수처럼 다룰 수 있다.

✓ 함수는 일급 객체

code

```
function add(a, b) {  
  return a + b  
}  
  
// 함수를 다른 함수의 인자로 넘긴다.  
[1, 2, 3].reduce(add, 0)  
  
(() => {  
  console.log('익명 함수를 생성한다.')})();  
  
function outer(a) {  
  function inner(b) {  
    return a + b  
  } // 중첩 함수를 생성한다.  
  return inner(10)  
}
```

```
const Person = (name) => {  
  // 함수를 변수로 생성한다.  
  const printName = () => console.log(name)  
  return { printName }  
} // 함수를 리턴하며 closure를 생성한다.  
  
const person = Person('Daniel')  
person.printName()  
  
function printName(name) {  
  console.log('name : ', name)  
}  
  
// 함수끼리 비교한다.  
// === 의 경우, 변수가 같은 객체(함수)를 가리키는지 체크한다.  
console.log(printName === person.printName)
```

✓ 클로저

- 자바스크립트 클로저는, 함수의 일급 객체 성질을 이용한다.
- 함수가 생성될 때, 함수 내부에서 사용되는 변수들이 외부에 존재하는 경우 그 변수들은 함수의 스코프에 저장된다.
- 함수와 함수가 사용하는 변수들을 저장한 공간을 클로저(closure)라 한다.

✓ 클로저

createCard.js

```
function createCard() {  
  let title = "";  
  let content = "";  
  function changeTitle(text) {title =  
text}  
  function changeContent(text) {content =  
text}  
  function print() {  
    console.log("TITLE - ", title);  
    console.log("CONTENT - ", content);  
  }  
  return { changeTitle, changeContent,  
print };  
}
```

```
const card1 = createCard();  
  
card1.changeTitle("생일카드");  
card1.changeContent("생일축하해");  
card1.print();  
  
const card2 = createCard();  
  
card2.changeTitle("감사카드");  
card2.changeContent("고마워");  
card2.print();
```

✓ 클로저

code

```
let rate = 1.05;

function app() {
  let base = 10;

  return function (price) {
    return price * rate + base;
  };
}

const getPrice = app();
getPrice(120) // 136
```

- base는 app 함수 내부, rate는 app 함수 내부의 스코프에 존재한다.
- 함수가 참조하는 변수는 실행 시점에 실행 컨텍스트에 의해 스코프가 결정된다.

✓ 클로저

code

```
let rate = 1.05;

function app() {
  let base = 10;

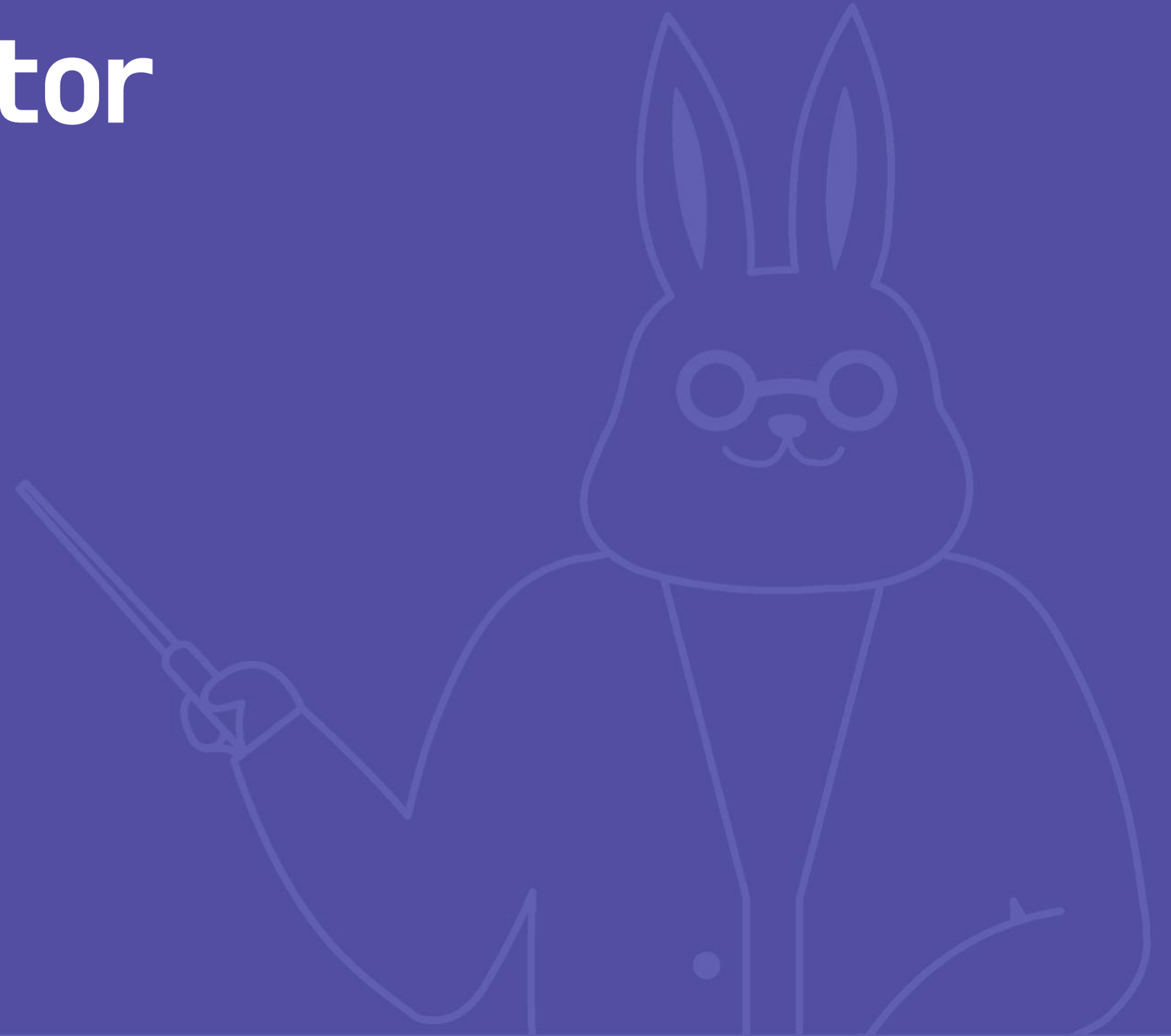
  return function (price) {
    return price * rate + base;
  };
}

console.log(app()(1)); // 11.05
rate = 1.1;
console.log(app()(1)); // 11.1
```

- 스코프에 따라서 변수에 영향을 받는다.
- rate의 변경은 두 클로저 함수 호출에 반영되지만, base는 증가해도 영향을 미치지 않는다.
- base는 app 호출 시 매번 생성되는 반면, rate는 매번 생성되지 않는다.

06

ES6 Rest, Spread Operator



✓ Rest Operator

- 함수의 인자, 배열, 객체 중 나머지 값을 묶어 사용하도록 한다.
- 함수의 인자 중 나머지를 가리킨다.
- 배열의 나머지 인자를 가리킨다.
- 객체의 나머지 필드를 가리킨다.

✓ 함수 인자 Rest Operator

code

```
function findMin(...rest) {  
  return rest.reduce((a, b) =>  
    a < b ? a : b)  
}
```

```
findMin(7, 3, 5, 2, 4, 1) // 1
```

- 함수 인자 rest operator는, 인자들을 배열로 묶는다.
- rest에는 숫자들이 배열로 담긴다.
- reduce 함수로 min 값을 리턴한다.

✓ 객체 Rest Operator

code

```
const o = {  
  name: "Daniel",  
  age: 23,  
  address: "Street",  
  job: "Software Engineer",  
};  
  
const { age, name, ...rest } = o;  
findSamePerson(age, name);
```

- 객체의 rest operator는, 지정된 필드 외의 나머지 필드를 객체로 묶는다.
- age, name을 제외한 나머지 필드는, rest 변수로 할당된다.

✓ 배열 Rest Operator

code

```
function sumArray(sum, arr) {  
  if (arr.length === 0) return sum;  
  const [head, ...tail] = arr;  
  return sumArray(sum + head, tail);  
}
```

```
sumArray(0, [1, 2, 3, 4, 5]);
```

- 배열의 rest operator는 나머지 인자를 다시 배열로 묶는다.
- sumArray의 tail 변수는, 첫 번째 원소 head를 제외한 나머지 값들을 다시 배열로 묶는다.
- tail은 하나씩 줄어들게 되며, 길이가 0이 되면 합을 반환한다.

✓ Spread Operator

- 묶인 배열 혹은 객체를 각각의 필드로 변환한다.
- 객체는 또 다른 객체로의 spread를 지원한다.
- 배열은 또 다른 배열의 인자, 함수의 인자로의 spread를 지원한다.

✓ 객체 Spread Operator

code

```
let o = {
  name: "Daniel",
  age: 23,
  address: "Street",
  job: "Software Engineer",
}

let o2 = { ...o, name: "Tom", age:
24 }

let o3 = { name: "Tom", age: 24,
...o }

o2.job // Software Engineer
o3.name // Daniel
```

- spread operator의 등장 순서에 따라, 객체의 필드가 덮어쓰워 질 수 있다.
- ...o 가 뒤에 등장하면, 기존의 name 필드가 나중에 등장하여 앞의 name: "Tom"을 덮어쓰운다.

✓ 배열 Spread operator

code

```
function findMinInObject(o) {  
  return Math.min(  
    ...Object.values(o)  
  )  
}
```

```
let o1 = { a: 1 }  
let o2 = { b: 3 }  
let o3 = { c: 7 }
```

```
findMinInFields(  
  mergeObjects(o1, o2, o3)  
) // 1
```

- mergeObjects는 주어진 객체들의 필드를 합친다.
- findMinInObject에서는 객체의 필드들 중 최솟값을 반환한다.
- Object.values는 객체 값들의 배열을 반환한다.
- 배열 spread operator로, Math.min의 인자를 넘긴다.

크레딧

/* elice */

코스 매니저

이재성

콘텐츠 제작자

김일식

강사

김일식

감수자

김일식

디자이너

김루미

연락처

TEL

070-4633-2015

WEB

<https://elice.io>

E-MAIL

contact@elice.io

