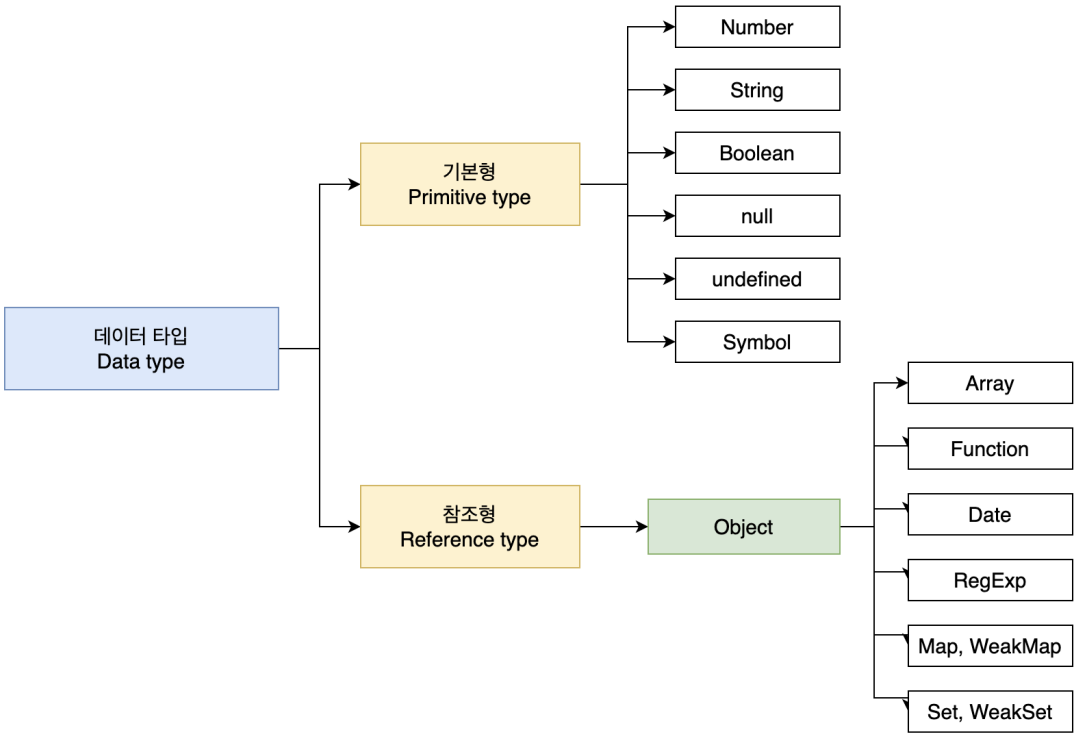
3주차 : 데이터 타입(심화), 실행 컨텍스트, this

[학습 목표]

1. 자바스크립트의 데이터 타입과 특징, 사용 시 주의할 점을 이해합니다.
2. 변수가 메모리에 할당되는 과정과 메모리에 대한 기본 지식을 습득합니다.
3. 실행 컨텍스트와 구성 요소인 VariableEnvironment과 LexicalEnvironment의 역할과 차이점, 스코프 체인과 this에 대해 이해합니다.

1. 데이터 타입 심화

▼ (1) 데이터 타입의 종류(기본형과 참조형)



(이미지 출처 : https://velog.io/@imjkim49/자바스크립트-데이터-타입-정리)

자바스크립트에서 값의 타입은 크게 기본형(Primitive Type)과 참조형(Reference Type)으로 구분됩니다. 기본형과 참조형의 구분 기준은 `값의 저장 방식` **과,** `불변성 여부` 입니다.

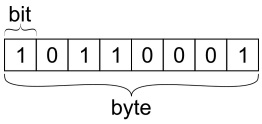
💡 **[기본형과 참조형의 구분 기준]**

1. **복제의 방식**
    a. 기본형 : 값이 담긴 주소값을 바로 복제
    b. 참조형 : 값이 담긴 주소값들로 이루어진 묶음을 가리키는 주소값을 복제
2. **불변성**의 여부
    a. 기본형 : 불변성을 띔
    b. 참조형 : 불변성을 띄지 않음

자, "불변성을 띈다" 이 말을 이해하기 위해서 우리는 메모리와 데이터에 대한 내용을 이해해야만 합니다. 아래에서 그 배경지식을 낱낱이 살펴보기로 합시다 😎

▼ (2) 메모리와 데이터에 관한 배경지식

1. 메모리, 데이터
    a. 비트
        i. 컴퓨터가 이해할 수 있는 가장 작은 단위죠
        ii. 0과 1을 가지고 있는 **메모리를 구성하기 위한 작은 조각**을 의미한다고 보면 될 것 같아요!
        iii. 이 작은 조각들이 모여서 여러분들이 흔히 들으시는 **'메모리'**가 만들어지는 것이죠.
    b. 바이트
        i. 0과 1만 표현하는 비트를 모두 찾기는 부담
        ii. 1개 → 2개 → … → 8개(새로운 단위 : byte)



(이미지 출처 : https://namu.wiki/w/바이트)

    c. 메모리(memo + ry) : byte 단위로 구성
        i. 모든 데이터는 byte 단위의 식별자인 메모리 주소값을 통해서 서로 구분이 됩니다.

💡 **만일, 64비트(8바이트) 정수는 메모리에 어떻게 저장할 수 있을까요?**
⇒ 64비트를 8개의 바이트로 분할하고, 각 바이트를 메모리에 저장해야 해요. 각 바이트는 8개의 비트를 가므로 64비트 정수는 메모리에서 **8개의 연속된 바이트**에 저장된답니다.

    d. java, c와 다른 javascript의 메모리 관리 방식(feat. 정수형)
        i. 8을 저장하는 방법
            1. JS : let a = 8(8byte)
            2. JAVA
                a. byte a = 8(1byte)
                b. short a = 8(2byte)
                c. int a = 8(4byte)
                d. long a = 8(16byte)
        ii. java 또는 c언어가 초기에 등장했을 때 숫자 데이터 타입은 크기에 따라 다양하게 지정해줘야 할 만큼 개발자가 **handling 할 요소**들이 많았어요. 하지만 javascript는 이런 부분에서는 상당히 편리하죠. 메모리 이슈까지는 고민하지 않아도 되니까요 😎

2. 식별자, 변수
    a. var testValue = 3
    b. 변수 = 데이터
    c. 식별자 = 변수명

▼ (3) 변수 선언과 데이터 할당

1. 할당 예시(풀어 쓴 방식, 붙여 쓴 방식 동일하게 동작해요)

```
/** 선언과 할당을 풀어 쓴 방식 */
var str;
str = 'test!';

/** 선언과 할당을 붙여 쓴 방식 */
var str = 'test!';
```

⬇️ 강의를 참고하셔서, 직접 아래 표를 작성해보세요!

주소	…	1002	1003	1004	1005	…
데이터						
주소	…	5002	5003	5004	5005	…
데이터						

2. 값을 바로 변수에 대입하지 않는 이유(=무조건 새로 만드는 이유)
    a. 자유로운 데이터 변환
        i. 이미 입력한 문자열이 길어진다면?
        ii. 숫자는 항상 8byte로 고정이지만, 문자는 고정이 아니에요(영문 : 1byte, 한글 : 2byte). 그래서, 이미 **1003 주소에 할당된 데이터**를 변환하려 할 때 훨씬 더 큰 데이터를 저장하려 한다면 → 1004 이후부터 저장되어있는 모든 데이터를 오른쪽으로 다~~~ 미뤄야 하겠죠..!?
    b. 메모리의 효율적 관리
        i. 똑같은 데이터를 여러번 저장해야 한다면?
        ii. 1만개의 변수를 생성해서 모든 변수에 숫자 1을 할당하는 상황을 가정해 봅시다. 모든 변수를 별개로 인식한다고 한다면, 1만개의 변수 공간을 확보해야 해요.
            1. 바로 대입하는 case) 숫자형은 8 바이트 고정이죠?
                a. 1만개 * 8byte = **8만 byte**
            2. 변수 영역에 별도 저장 case)
                a. 변수 영역 : 2바이트 1만개 = 2만바이트

> ℹ️ 이해를 돕고자, 변수 영역에 저장되는 데이터는 2바이트로 가정했어요!

                b. 데이터 영역 : 8바이트 1개 = 8바이트
                c. 총 : **2만 8바이트**

▼ (4) 기본형 데이터와 참조형 데이터

1. 메모리를 기준으로 다시한번 생각해보는 **두 가지 주요 개념**
    a. 변수 vs 상수
        i. 변수 : 변수 영역 메모리를 변경할 수 있음
        ii. 상수 : 변수 영역 메모리를 변경할 수 없음
    b. 불변하다 vs 불변하지 않다
        i. 불변하다 : 데이터 영역 메모리를 변경할 수 없음

ii. 불변하지 않다 : 데이터 영역 메모리를 변경할 수 있음

2. 불변값과 불변성(with 기본형 데이터)

```
// a라는 변수가 abc에서 abcdef가 되는 과정을 통해 불변성을 유추해봅시다!

// 'abc'라는 값이 데이터영역의 @5002라는 주소에 들어갔다고 가정할게요.
var a = 'abc';

// 'def'라는 값이 @5002라는 주소에 추가되는 것이 아니죠!
// @5003에 별도로 'abcdef'라는 값이 생기고 a라는 변수는 @5002 -> @5003
// 즉, "변수 a는 불변하다." 라고 할 수 있습니다.
// 이 때, @5002는 더 이상 사용되지 않기 때문에 가비지컬렉터의 수거 대상이 됩니다.
a = a + 'def';
```

3. 가변값과 가변성(with 참조형 데이터)

a. 참조형 데이터의 변수 할당 과정

```
// 참조형 데이터는 별도 저장공간(obj1을 위한 별도 공간)이 필요합니다!
var obj1 = {
  a: 1,
  b: 'bbb,
};
```

⬇️ 강의를 참고해서 아래 표를 직접 작성해보세요!

주소	1001	1002	1003	1004
데이터				
주소	5001	5002	5003	5004
데이터				

주소	7103	7104	7105	7106
데이터				

b. 기본형 데이터의 변수 할당 과정과 차이점 : **객체의 변수(프로퍼티) 영역**의 별도 존재 여부

c. 참조형 데이터가 불변하지 않다(가변하다)라고 하는 이유

```
var obj1 = {
  a: 1,
  b: 'bbb',
};

// 데이터를 변경해봅시다.
obj1.a = 2;
```

1. 위에 작성해놓았던 표를 토대로 변경 테스트를 해보며 알아봅시다 😉
2. 과정은 아래와 같아요.
   a. 변경할 값인 숫자 2를 데이터 영역에서 검색합니다.
   b. 없네요! 2를 새로 추가하고, 해당 주소(ex : @5003)를 obj1을 위한 별도 영역에 갈아껴줍니다.
3. 데이터 영역에 저장된 값은 여전히 계속 불변값이지만, obj1을 위한 별도 영역은 얼마든지 변경이 가능해요. 이것 때문에 참조형 데이터를 흔히, `'불변하지 않다(=가변하다)'` 라고 합니다.

d. 중첩객체의 할당

> 자바스크립트에서 중첩객체란, 객체 안에 또 다른 객체가 들어가는 것을 말해요. 이번 주차 초반에 살펴보았듯이 객체는 배열, 함수 등을 모두 포함하는 상위개념이기 때문에 **배열을 포함하는 객체도 중첩객체**라고 할 수 있답니다.

```
var obj = {
  x: 3,
  arr: [3, 4, 5],
```

```
}

// obj.arr[1]의 탐색과정은 어떻게 될까요? 작성하신 표에서 한번 찾아가보세요!
```

⬇️ 강의를 참고해서 아래 표를 직접 작성해보세요!

주소	1001	1002	1003	1004	1005	…
데이터						
주소	5001	5002	5003	5004	5005	…
데이터						

주소	7103	7104	…
데이터			

주소	8104	8105	8106	…

e. 참조 카운트가 0인 메모리 주소의 처리

i. 참조카운트란 무엇일까요?

> 💡 **객체를 참조하는 변수나 다른 객체의 수를 나타내는 값**입니다. 참조 카운트가 0인 객체는 더 이상 사용되지 않으므로, **가비지 컬렉터**에 의해 메모리에서 제거됩니다.

ii. 가비지컬렉터(GC, Garbage Collector)

> 💡 더 이상 사용되지 않는 객체를 자동으로 메모리에서 제거하는 역할을 합니다. 자바스크립트는 가비지 컬렉션을 수행함으로써 개발자가 명시적으로 메모리 관리를 하지 않아도 되도록 지원합니다. 자바스크립트 엔진에서 내부적으로 수행되며, 개발자는 가비지 컬렉션에 대한 직접적인 제어를 할 수 없습니다.

4. 변수 복사의 비교

우리가 지금까지 작성했던 방법과 상당히 유사합니다. 먼저 기본형 및 참조형 데이터의 선언과 할당을 해보고, 변수 복사까지 진행해봅시다! 강의를 참고하여 역시 표를 작성해주세요 😎

```
// STEP01. 쭉 선언을 먼저 해볼께요.
var a = 10; //기본형
var obj1 = { c: 10, d: 'ddd' }; //참조형

// STEP02. 복사를 수행해볼께요.
var b = a; //기본형
var obj2 = obj1; //참조형
```

주소	1001	1002	1003	1004	1005	…
데이터						
주소	5001	5002	5003	5004	5005	
데이터						

주소	7103	7104	…
데이터			

5. 복사 이후 값 변경(객체의 프로퍼티 변경)

자, 복사까지는 할만 하셨죠? 기본형과 참조형의 두드러지는 차이는 복사한 후의 값 변경에서 일어나요. 한번 수행하고 표를 작성해 보시면서 어떤 차이가 있는지 찾아보도록 합시다 🔥
(값 변경 부분에 대한 작성도 위의 표에 해주세요!)

```
// STEP01. 쭉 선언을 먼저 해볼께요.
var a = 10; //기본형
var obj1 = { c: 10, d: 'ddd' }; //참조형
```

```
// STEP02. 복사를 수행해볼께요.
var b = a; //기본형
var obj2 = obj1; //참조형

b = 15;
obj2.c = 20;
```

기본형과 참조형의 변수 복사 시 주요한 절차의 차이점은 다음과 같아요!

- 기본형
  - 숫자 15라는 값을 데이터 영역에서 검색 후 없다면 생성
  - 검색한 결과주소 또는 생성한 주소를 변수 영역 b에 갈아끼움
  - a와 b는 서로 다른 데이터 영역의 주소를 바라보고 있기 때문에 `영향 없음`
- 참조형
  - 숫자 20이라는 값을 데이터 영역에서 검색 후 없다면 생성
  - 검색한 결과주소 또는 생성한 주소 obj2에게 지정되어 있는 별도 영역(7103~)에 갈아끼움
  - obj1도 똑같은 주소를 바라보고 있기 때문에 `obj1까지 변경이 됨`
  - 바로 아래와 같은 현상이 생기는 것이죠!

```
// 기본형 변수 복사의 결과는 다른 값!
a !== b;

// 참조형 변수 복사의 결과는 같은 값!(원하지 않았던 결과😭)
obj1 === obj2;
```

6. 복사 이후 값 변경(객체 자체를 변경)

   만약, 객체의 프로퍼티(=속성)에 접근해서 값을 변경하는 것이 아니라 객체 자체를 변경하는 방식으로 값을 바꾼다면 어떨까요? 아래 예제를 한번 같이 따라가볼까요? 🚀

```
//기본형 데이터
var a = 10;
var b = a;

//참조형 데이터
var obj1 = { c: 10, d: 'ddd' };
var obj2 = obj1;

b = 15;
obj2 = { c: 20, d: 'ddd'};
```

⬇️ 강의를 참고해서 아래 표를 직접 작성해보세요!

주소	1001	1002	1003	1004	1005	1006
데이터						
주소	5001	5002	5003	5004	5005	5006
데이터						

주소	7103	7104	…
데이터			

주소	8204	8205	…
데이터			

> obj2 변수는 참조형 데이터이고, 참조형 데이터의 값을 변경한 것임에도 불고하고 이전 케이스와는 다르게 obj1과는 바라보는 데이터 메모리 영역의 값이 달라졌습니다!

참조형 데이터가 '가변값'이라고 할 때의 '가변'은 참조형 데이터 자체를 변경할 경우가 아니라, 그 내부의 프로퍼티를 변경할 때 성립한다고 할 수 있겠네요 👍

▼ (5) 불변 객체

1. 불변 객체의 정의

   우리는 앞선 과정에서, `가변하다`와 `불변하다`의 개념을 배웠습니다. 다시 살짝 정리해서 객체를 예로 들면, 객체의 속성에 접근해서 값을 변경하면 ***가변이 성립***했었죠. 반면, 객체 데이터 자체를 변경(새로운 데이터를 할당)하고자 한다면 기존 데이터는 변경되지 않습니다. 즉, ***불변하다라고*** 볼 수 있습니다.

   개념은 알겠는데, 그 개념이 왜 필요한지 알겠나요? 아래 예시를 통해서 "불변하다". 혹은, '불변객체'의 개념이 왜 필요한지를 한번 알아보도록 합시다 🐱

2. 불변 객체의 필요성

   a. 다음 예시는 객체의 가변성에 따른 문제점을 보여주고 있어요 🙄

```
// user 객체를 생성
var user = {
  name: 'wonjang',
  gender: 'male',
};

// 이름을 변경하는 함수, 'changeName'을 정의
// 입력값 : 변경대상 user 객체, 변경하고자 하는 이름
// 출력값 : 새로운 user 객체
// 특징 : 객체의 프로퍼티(속성)에 접근해서 이름을 변경했네요! -> 가변
var changeName = function (user, newName) {
  var newUser = user;
  newUser.name = newName;
  return newUser;
};

// 변경한 user정보를 user2 변수에 할당하겠습니다.
// 가변이기 때문에 user1도 영향을 받게 될거에요.
var user2 = changeName(user, 'twojang');

// 결국 아래 로직은 skip하게 될겁니다.
if (user !== user2) {
  console.log('유저 정보가 변경되었습니다.');
}

console.log(user.name, user2.name); // twojang twojang
console.log(user === user2); // true
```

   b. 위의 예제를 아래와 같이 개선할 수 있어요!

```
// user 객체를 생성
var user = {
  name: 'wonjang',
  gender: 'male',
};

// 이름을 변경하는 함수 정의
// 입력값 : 변경대상 user 객체, 변경하고자 하는 이름
// 출력값 : 새로운 user 객체
// 특징 : 객체의 프로퍼티에 접근하는 것이 아니라, 아에 새로운 객체를 반환 -> 불변
var changeName = function (user, newName) {
  return {
    name: newName,
    gender: user.gender,
  };
};

// 변경한 user정보를 user2 변수에 할당하겠습니다.
// 불변이기 때문에 user1은 영향이 없어요!
var user2 = changeName(user, 'twojang');

// 결국 아래 로직이 수행되겠네요.
if (user !== user2) {
  console.log('유저 정보가 변경되었습니다.');
}

console.log(user.name, user2.name); // wonjang twojang
console.log(user === user2); // false 👍
```

   c. 위 방법이 과연 최선일까요?

   안타깝지만, 그렇진 않아요. 다음과 같은 문제점이 있기 때문입니다.

- changeName 함수는 새로운 객체를 만들기 위해 변경할 필요가 없는 gender 프로퍼티를 하드코딩으로 입력했어요 ⇒ 만일 이러한 속성이 10개라면? 🥲
- 따라서, 다음 제시하는 `얕은 복사`의 방법을 제시할게요!

3. 더 나은 방법 : 얕은 복사
   a. 패턴과 적용

```
//이런 패턴은 어떨까요?
var copyObject = function (target) {
  var result = {};

  // for ~ in 구문을 이용하여, 객체의 모든 프로퍼티에 접근할 수 있습니다.
  // 하드코딩을 하지 않아도 괜찮아요.
  // 이 copyObject로 복사를 한 다음, 복사를 완료한 객체의 프로퍼티를 변경하면
  // 되겠죠!?
  for (var prop in target) {
    result[prop] = target[prop];
  }
  return result;
}
```

```
//위 패턴을 우리 예제에 적용해봅시다.
var user = {
  name: 'wonjang',
  gender: 'male',
};

var user2 = copyObject(user);
user2.name = 'twojang';

if (user !== user2) {
  console.log('유저 정보가 변경되었습니다.');
}

console.log(user.name, user2.name);
console.log(user === user2);
```

   b.

4. 얕은 복사 vs 깊은 복사

   ✓ 이 패턴도 여전히 문제가 있을까요?

   하하….. 네 ㅠㅠ 여전히 있습니다. 왜냐면, 중첩된 객체에 대해서는 **완벽한 복사를 할 수 없기 때문**이에요. 이것이 얕은 복사의 한계입니다.

   1. 얕은 복사 : 바로 아래 단계의 값만 복사(위의 예제)
      문제점 : 중첩된 객체의 경우 참조형 데이터가 저장된 프로퍼티를 복사할 때, 주소값만 복사
   2. 깊은 복사 : 내부의 모든 값들을 하나하나 다 찾아서 모두 복사하는 방법
   3. 중첩된 객체에 대한 얕은 복사 살펴보기

```
var user = {
  name: 'wonjang',
  urls: {
    portfolio: 'http://github.com/abc',
    blog: 'http://blog.com',
    facebook: 'http://facebook.com/abc',
  }
};

var user2 = copyObject(user);

user2.name = 'twojang';

// 바로 아래 단계에 대해서는 불변성을 유지하기 때문에 값이 달라지죠.
console.log(user.name === user2.name); // false

// 더 깊은 단계에 대해서는 불변성을 유지하지 못하기 때문에 값이 같아요.
// 더 혼란스러워 지는거죠 ㅠㅠ
user.urls.portfolio = 'http://portfolio.com';
console.log(user.urls.portfolio === user2.urls.portfolio); // true

// 아래 예도 똑같아요.
```

```
user2.urls.blog = '';
console.log(user.urls.blog === user2.urls.blog); // true
```

4. **결국, ser.urls 프로퍼티도 불변 객체로 만들어야 해요.**
5. 중첩된 객체에 대한 깊은 복사 살펴보기

```
var user = {
  name: 'wonjang',
  urls: {
    portfolio: 'http://github.com/abc',
    blog: 'http://blog.com',
    facebook: 'http://facebook.com/abc',
  }
};

// 1차 copy
var user2 = copyObject(user);

// 2차 copy -> 이렇게까지 해줘야만 해요..!!
user2.urls = copyObject(user.urls);

user.urls.portfolio = 'http://portfolio.com';
console.log(user.urls.portfolio === user2.urls.portfolio);

user2.urls.blog = '';
console.log(user.urls.blog === user2.urls.blog);
```

6. 결론 : 객체의 프로퍼티 중, 기본형 데이터는 그대로 복사 + 참조형 데이터는 다시 그 내부의 프로퍼티를 복사 ⇒ **재귀적 수행!**

> 💡 **재귀적으로 수행한다?**
> ⇒함수나 알고리즘이 자기 자신을 호출하여 반복적으로 실행되는 것을 말합니다 😎

7. [결론]을 적용한 코드 —> 완벽히 다른 객체를 반환하네요.

```
var copyObjectDeep = function(target) {
  var result = {};
  if (typeof target === 'object' && target !== null) {
    for (var prop in target) {
      result[pop] = copyObjectDeep(target[prop]);
    }
  } else {
    result = target;
  }
  return result;
}
```

이렇게 되면, 우리가 그토록 원하던 **'깊은 복사'를 완벽하게** 구현할 수 있습니다.

```
//결과 확인
var obj = {
  a: 1,
  b: {
    c: null,
    d: [1, 2],
  }
};
var obj2 = copyObjectDeep(obj);

obj2.a = 3;
obj2.b.c = 4;
obj2.b.d[1] = 3;

console.log(obj);
console.log(obj2);
```

8. 마지막 방법! JSON(=**JavaScript Object Notation)을 이용하는 방법**도 존재합니다. 하지만 완벽한 방법은 아니에요. 간략히 장/단점을 정리해드리니, 내용만 참고해주세요 😉

   장점:

- JSON.stringify() 함수를 사용하여 객체를 문자열로 변환한 후, 다시 JSON.parse() 함수를 사용하여 새로운 객체를 생성하기 때문에, 원본 객체와 복사본 객체가 서로 독립적으로 존재합니다. 따라서 복사본 객체를 수정해도 원본 객체에 영향을 미치지 않습니다.
- JSON을 이용한 깊은 복사는 다른 깊은 복사 방법에 비해 코드가 간결하고 쉽게 이해할 수 있습니다.

단점:

- JSON을 이용한 깊은 복사는 원본 객체가 가지고 있는 모든 정보를 복사하지 않습니다. 예를 들어, 함수나 undefined와 같은 속성 값은 복사되지 않습니다.
- JSON.stringify() 함수는 순환 참조(Recursive Reference)를 지원하지 않습니다. 따라서 객체 안에 객체가 중첩되어 있는 경우, 이 방법으로는 복사할 수 없습니다.

따라서 JSON을 이용한 깊은 복사는 객체의 구조가 간단하고, 함수나 undefined와 같은 속성 값이 없는 경우에 적합한 방법입니다. 만약 객체의 구조가 복잡하거나 순환 참조가 있는 경우에는 다른 깊은 복사 방법을 고려해야 합니다.

▼ (6) undefined와 null

둘 다 없음을 나타내는 값이에요. 하지만 미세하게 다르고, 그 목적 또한 다르답니다. 아래에서 한번 살펴보도록 할께요!

1. undefined
   a. 사용자(=개발자)가 직접 지정할 수도 있지만 일반적으로는 자바스크립트 엔진에서 **값이 있어야 할 것 같은데 없는 경우**, 자동으로 부여합니다. 다음 케이스를 생각해 주시면 될 것 같아요.
      i. 변수에 값이 지정되지 않은 경우, 데이터 영역의 메모리 주소를 지정하지 않은 식별자에 접근할 때
      ii. .이나 []로 접근하려 할 때, 해당 데이터가 존재하지 않는 경우
      iii. return 문이 없거나 호출되지 않는 함수의 실행 결과

```
var a;
console.log(a); // (1) 값을 대입하지 않은 변수에 접근

var obj = { a: 1 };
console.log(obj.a); // 1
console.log(obj.b); // (2) 존재하지 않는 property에 접근
// console.log(b); // 오류 발생

var func = function() { };
var c = func(); // (3) 반환 값이 없는 function
console.log(c); // undefined
```

   b. 2가지 역할을 가진 undefined, 헷갈릴 만도 해요. 그리고 위험해요!
      i. 지금 undefined로 나오는 이 변수가, 필요에 의해 할당한건지 자바스크립트 엔진이 반환한건지 어떻게 알죠? → 구분할 수 없어요.
      ii. '없다'를 명시적으로 표현할 때는 undefined를 사용하지 맙시다!
2. null
   a. 용도 : '없다'를 명시적으로 표현할 때
   b. 주의 : typeof null

      typeof null이 object인 것은 유명한 javascript 자체 버그입니다. 조심해야겠죠?

```
var n = null;
console.log(typeof n); // object

//동등연산자(equality operator)
console.log(n == undefined); // true
console.log(n == null); // true

//일치연산자(identity operator)
console.log(n === undefined);
console.log(n === null);
```

2. 실행컨텍스트(스코프, 변수, 객체, 호이스팅)

자바스크립트의 `실행 컨텍스트`는 실행할 코드에 제공할 **환경 정보**들을 모아놓은 **객체**입니다. 자바스크립트는 어떤 실행 컨텍스트가 활성화 되는 시점에 다음과 같은 일을 합니다.

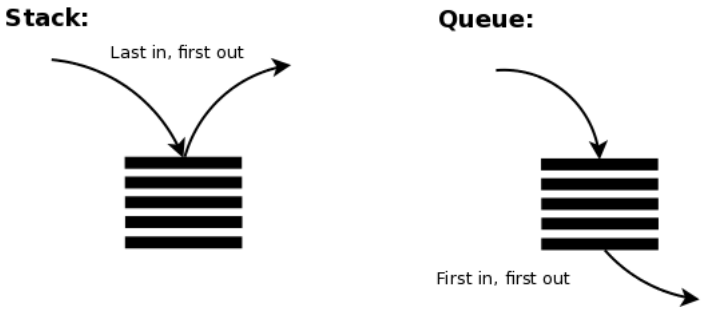
1. 선언된 변수를 위로 끌어올리구요 = 호이스팅(hoisting)
2. 외부 환경 정보를 구성하구요.
3. this 값을 설정해요.

이런 현상들 때문에 **JS에서는 다른 언어랑은 다른 특징**들이 나타난답니다.

▼ (1) 실행 컨텍스트란?

실행 컨텍스트를 이해하기 위해서는, `콜 스택`에 대한 이해가 반드시 필요합니다. 자, 그 전에 `"스택"`이라는 개념에 대해서 먼저 이해를 해야겠네요.

▼ 스택 vs 큐



(이미지 출처 : https://velog.io/@leejuhwan/스택STACK과-큐QUEUE)

▼ 콜 스택(call stack)

실행 컨텍스트란 **실행할 코드에 제공할 환경 정보**들을 모아놓은 객체라고 했었죠. 그 객체. 즉, 동일 환경에 있는 코드를 실행할 때 필요한 환경 정보들을 모아 컨텍스트를 구성하고 이것을 위에서 설명드린 '스택'의 한 종류인 `콜스택`에 쌓아올립니다. 가장 위에 쌓여있는 컨텍스트와 관련된 코드를 실행하는 방법으로 코드의 환경 및 순서를 보장할 수 있어요.

1. **컨텍스트의 구성**
   a. 구성방법(여러가지가 있지만, 사실 우리는 함수만 생각하면 돼요 👍)
      i. 전역공간
      ii. eval()함수
      iii. **함수(우리가 흔히 실행컨텍스트를 구성하는 방법)**
   b. 실행컨텍스트 구성 예시 코드

   ✓ 자 먼저, 실행컨텍스트에 대해 생각하기 전에 여러분이 눈으로 한번 어떻게 실행될지 예상해보시겠어요? 👀

```
// ---- 1번
var a = 1;
function outer() {
  function inner() {
    console.log(a); //undefined
    var a = 3;
  }
  inner(); // ---- 2번
  console.log(a);
}
outer(); // ---- 3번
console.log(a);
```

   c. 실행컨텍스트 구성 순서

   위 코드는 아래 순서로 진행이 됩니다.
   (콜 스택에 쌓이는 실행컨텍스트에 의해 순서가 보장되니까요!)

   코드실행 → 전역(in) → 전역(중단) + outer(in) → outer(중단) + inner(in) → inner(out) + outer(재개) → outer(out) + 전역(재개) → 전역(out) → 코드종료

d. 결국은 특정 실행 컨텍스트가 생성되는(또는 활성화되는) 시점이 콜 스택의 맨 위에 쌓이는(노출되는) 순간을 의미하구요. 곧, 현재 실행할 코드에 해당 실행 컨텍스트가 관여하게 되는 시점을 의미한다고 받아들여주시면 정확합니다! 👍👍

▼ **실행 컨텍스트 객체의 실체(=담기는 정보)**

계속 이 객체에 대해 얘기는 해오고 있는데, 실체에 대해 감은 잘 안오시죠? 그 실체를 한번 알아봅시다. 3가지로 알아볼께요. 지금은 용어가 어렵지만, 앞으로 계속해서 언급할테니 너무 걱정마세요 😎

1. VariableEnvironment
   a. 현재 컨텍스트 내의 식별자 정보(=record)를 갖고있어요.
      i. `var a = 3`
      ii. 위의 경우, `var a`를 의미
   b. 외부 환경 정보(=outer)를 갖고있어요.
   c. 선언 시점 LexicalEnvironment의 **snapshot**
2. LexicalEnvironment
   a. VariableEnvironment와 동일하지만, 변경사항을 실시간으로 반영해요.
3. ThisBinding
   a. this 식별자가 바라봐야할 객체

▼ (2) VariableEnvironment, LexicalEnvironment의 개요

▼ **VE vs LE**

이 두가지는 담기는 항목은 완벽하게 동일해요. 그러나, 스냅샷 유지여부는 다음과 같이 달라요.

1. VE : 스냅샷을 유지해요.
2. LE : 스냅샷을 유지하지 않아요. 즉, **실시간으로 변경사항을 계속해서 반영**합니다.

> 결국, 실행 컨텍스트를 생성할 때, VE에 정보를 먼저 담은 다음, 이를 그대로 복사해서 LE를 만들고 이후에는 주로 LE를 활용한답니다.

▼ **구성 요소(VE, LE 서로 같아요!)**

1. VE, LE모두 동일하며, ‘environmentRecord’와 ‘outerEnvironmentReference’로 구성
2. environmentRecord(=record)
   a. 현재 컨텍스트와 관련된 코드의 식별자 정보들이 저장돼요.
   b. 함수에 지정된 매개변수 식별자, 함수자체, var로 선언된 변수 식별자 등
3. outerEnvironmentReference(=outer)

▼ (3) LexicalEnvironment(1) - environmentRocord(=record)와 호이스팅

▼ **개요**

1. 현재 컨텍스트와 관련된 코드의 **식별자 정보**들이 저장(수집)돼요. `기록된다`라고 이해해보면, `record`라는 말과 일맥상통하죠?
2. 수집 대상 정보 : 함수에 지정된 매개변수 식별자, 함수 자체, var로 선언된 변수 식별자 등
3. 컨텍스트 내부를 처음부터 끝까지 **순서대로** 훑어가며 수집

> 💡 순서대로 **수집**한다고 했지, 코드가 **실행**된다고 하지는 않았습니다!

▼ **호이스팅**

1. 변수정보 수집을 모두 마쳤더라도 아직 실행 컨텍스트가 관여할 코드는 실행 전의 상태에요(JS 엔진은 코드 실행 전 이미 모든 변수정보를 알고 있는 것)
2. 변수 정보 수집 과정을 이해하기 쉽게 설명한 **‘가상 개념’**

> 💡 가상개념이라는 말은, 실제로는 그렇진 않더라도 사람이 이해하기 쉬운 말로 풀어 표현했다는 것을 의미해요 😄

▼ 호이스팅 규칙

1. **호이스팅 법칙 1** : 매개변수 및 변수는 선언부를 호이스팅 합니다.

   다음 주석에 딸려있는 **3가지 action point**에 따라 실습을 진행해보세요!

   <적용 전>

```
//action point 1 : 매개변수 다시 쓰기(JS 엔진은 똑같이 이해한다)
//action point 2 : 결과 예상하기
//action point 3 : hoisting 적용해본 후 결과를 다시 예상해보기

function a (x) {
  console.log(x);
  var x;
  console.log(x);
  var x = 2;
  console.log(x);
}
a(1);
```

   <매개변수 적용>

```
//action point 1 : 매개변수 다시 쓰기(JS 엔진은 똑같이 이해한다)
//action point 2 : 결과 예상하기
//action point 3 : hoisting 적용해본 후 결과를 다시 예상해보기

function a () {
  var x = 1;
  console.log(x);
  var x;
  console.log(x);
  var x = 2;
  console.log(x);
}
a(1);
```

   <호이스팅 적용>

```
//action point 1 : 매개변수 다시 쓰기(JS 엔진은 똑같이 이해한다)
//action point 2 : 결과 예상하기
//action point 3 : hoisting 적용해본 후 결과를 다시 예상해보기

function a () {
  var x;
  var x;
  var x;

  x = 1;
  console.log(x);
  console.log(x);
  x = 2;
  console.log(x);
}
a(1);
```

   자, 우리는 예상을

   > 1 → undefined → 2로 예상했지만

   실제로는

   > 1, 1, 2 라는 결과가 나왔네요

   호이스팅이라는 개념을 모르면 예측이 불가능한 어려운 결과입니다.

2. **호이스팅 법칙 2** : 함수 선언은 전체를 호이스팅합니다.
   마찬가지로, **2가지 action points**에 따라 진행해봅시다.

<적용 전>

```
//action point 1 : 결과 값 예상해보기
//action point 2 : hoisting 적용해본 후 결과를 다시 예상해보기

function a () {
  console.log(b);
  var b = 'bbb';
  console.log(b);
  function b() { }
  console.log(b);
}
a();
```

<호이스팅 적용>

```
//action point 1 : 결과 값 예상해보기
//action point 2 : hoisting 적용해본 후 결과를 다시 예상해보기

function a () {
  var b; // 변수 선언부 호이스팅
  function b() { } // 함수 선언은 전체를 호이스팅

  console.log(b);
  b = 'bbb'; // 변수의 할당부는 원래 자리에

  console.log(b);
  console.log(b);
}
a();
```

해석을 편하게 하기 위해서 **함수선언문을 함수 표현식으로** 바꿔볼게요!

```
//action point 1 : 결과 값 예상해보기
//action point 2 : hoisting 적용해본 후 결과를 다시 예상해보기

function a () {
  var b; // 변수 선언부 호이스팅
  var b = function b() { } // 함수 선언은 전체를 호이스팅

  console.log(b);
  b = 'bbb'; // 변수의 할당부는 원래 자리에

  console.log(b);
  console.log(b);
}
a();
```

이번에도 우리의 예상은 틀렸네요.

> 에러(또는 undefined), ‘bbb’, b함수

라고 나올 것 같았지만, 실제로는

> b함수, ‘bbb’, ‘bbb’

라는 결과가 나왔어요. 이 또한 호이스팅을 고려하지 않고는 결과를 예측하기가 매우 어려웠어요. 호이스팅을 다루는 김에, **함수의 정의방식 3가지**와 **주의해야 할 내용**을 살짝 짚고 넘어가 보도록 하겠습니다 😉

3. 함수 선언문, 함수 표현식
   a. 함수 정의의 3가지 방식

```
// 함수 선언문. 함수명 a가 곧 변수명
// function 정의부만 존재, 할당 명령이 없는 경우
function a () { /* ... */ }
a(); // 실행 ok

// 함수 표현식. 정의한 function을 별도 변수에 할당하는 경우
// (1) 익명함수표현식 : 변수명 b가 곧 변수명(일반적 case에요)
var b = function () { /* ... */ }
b(); // 실행 ok
```

```
// (2) 기명 함수 표현식 : 변수명은 c, 함수명은 d
// d()는 c() 안에서 재귀적으로 호출될 때만 사용 가능하므로 사용성에 대한 의문
var c = function d () { /* ... */ }
c(); // 실행 ok
d(); // 에러!
```

b. 주의해야 할 내용

i. 함수 선언문, 함수 표현식

자, 정신없이 달려오느라 내용이 정리가 안됐을 수 있어요. 다시 잠깐 교통정리를 해드릴께요! 지금 우리는 아래 내용을 하고 있어요 😎

> 💡 - **실행 컨텍스트**는 **실행할 코드에 제공할 환경 정보**들을 모아놓은 객체이다.
> - 그 객체 안에는 3가지가 존재한다.
> ✓ VariableEnvironment
> ✓ LexicalEnvironment
> ✓ ThisBindings
> - VE와 LE는 실행컨텍스트 생성 시점에 내용이 완전히 같고, 이후 스냅샷 유지 여부가 다르다.
> - LE는 다음 2가지 정보를 가지고 있다.
> ✓ record(=environmentRecord) ← **이 record의 수집과정이 hoisting**
> ✓ outer(=outerEnvironmentReference)
>
> 길을 잃지 마세요 여러분 😅😅

계속해서 가 봅시다. 함수 선언문과 표현식을 배웠으니, 실질적인 차이를 예시를 통해 배워볼께요.

```
console.log(sum(1, 2));
console.log(multiply(3, 4));

function sum (a, b) { // 함수 선언문 sum
  return a + b;
}

var multiply = function (a, b) { // 함수 표현식 multiply
  return a + b;
}
```

위에서 정리해드린대로, LE는 record와 outer를 수집하죠. 그 중, record를 수집하는 과정에서 hoisting이 일어나고, 우리가 익히 알고있는대로 위로 `쭉` 끌어올려본 결과를 다시 써보면 아래와 같겠네요.

```
// 함수 선언문은 전체를 hoisting
function sum (a, b) { // 함수 선언문 sum
  return a + b;
}

// 변수는 선언부만 hoisting

var multiply;

console.log(sum(1, 2));
console.log(multiply(3, 4));

multiply = function (a, b) { // 변수의 할당부는 원래 자리
  return a + b;
};
```

어떤가요? 눈으로 볼 때는 몰랐지만, 함수 선언문과 함수 표현식은 hoisting 과정에서 극명한 차이를 보입니다. 차이는 알겠지만 이게 왜 위험한지 모르겠다구요? 네. 그러면 아래 예를 통해 그 이유를 확인해보죠.

ii. **함수 선언문**을 주의해야 하는 이유

```
...

console.log(sum(3, 4));

// 함수 선언문으로 짠 코드
```

```
// 100번째 줄 : 시니어 개발자 코드(활용하는 곳 -> 200군데)
// hoisting에 의해 함수 전체가 위로 쭉!
function sum (x, y) {
  return x + y;
}

...
...

var a = sum(1, 2);

...

// 함수 선언문으로 짠 코드
// 5000번째 줄 : 신입이 개발자 코드(활용하는 곳 -> 10군데)
// hoisting에 의해 함수 전체가 위로 쭉!
function sum (x, y) {
  return x + ' + ' + y + ' = ' + (x + y);
}

...

var c = sum(1, 2);

console.log(c);
```

iii. 만약 함수 표현식이었다면…?

```
...

console.log(sum(3, 4));

// 함수 표현식으로 짠 코드
// 함수 선언부만 위로 쭉!
// 이 이후부터의 코드만 영향을 받아요!
var sum = function (x, y) {
  return x + y;
}

...
...

var a = sum(1, 2);

...

// 함수 표현식으로 짠 코드
// 함수 선언부만 위로 쭉!
// 이 이후부터의 코드만 영향을 받아요!
var sum = function (x, y) {
  return x + ' + ' + y + ' = ' + (x + y);
}

...

var c = sum(1, 2);

console.log(c);
```

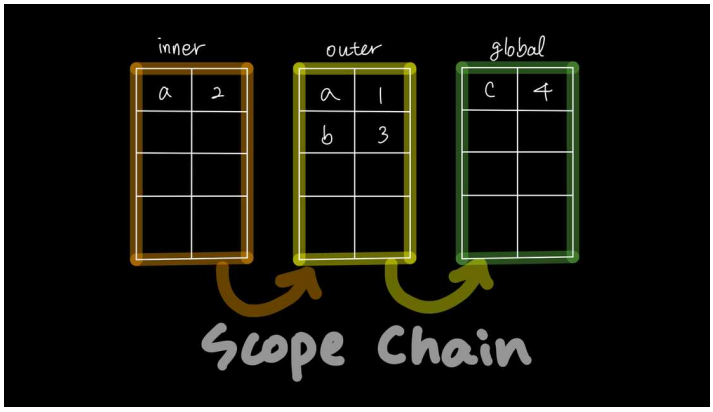
협업을 많이 하고, 복잡한 코드일 수록. 전역 공간에서 이루어지는 코드 협업일 수록 `함수 표현식`을 활용하는 습관을 들이도록 합시다!!

▼ (4) LexicalEnvironment(2) - 스코프, 스코프 체인, outerEnvironmentReference(=outer)

우리는 이미 앞서 '스코프'라는 용어에 대해 몇차례 언급을 해 왔습니다. **실행컨텍스트 관점에서의 스코프**를 같이 이해해 보도록 합시다.

▼ 주요 용어

1. 스코프
   a. 식별자에 대한 유효범위를 의미해요
   b. 대부분 언어에서 존재하구요, 당연하게도 JS에서도 존재하죠
2. 스코프 체인
   a. 식별자의 유효범위를 안에서부터 바깥으로 차례로 검색해나가는 것



출처 : https://jess2.xyz/JavaScript/scope-chain-closure/

3. outerEnvironmentReference(이하 outer)

   우리는 지금까지 LE의 구성요소 record와 outer 중 record에 대해서 깊이 알아오고 있었어요. 이번 시간의 주인공은 **그 두번째인 outer**입니다. outer의 역할을 한 마디로 정의하자면

   > **스코프 체인이 가능토록 하는 것(외부 환경의 참조정보)라고 할 수 있어요**
   > * 외부 환경의 참조정보 라는 말에 집중해주세요.

   위에 나온 개념을 좀 더 자세히 살펴보도록 하겠습니다.

▼ **스코프 체인**

1. **outer는 현재 호출된 함수가 선언될 당시**(이 말이 중요해요!)의 LexicalEnvironment를 참조해요.
   참조한다는 말이 어려우면, 그 당시의 환경 정보를 저장한다. 정보로 이해해도 괜찮습니다.
2. 예를 들어, `A함수 내부에 B함수 선언 → B함수 내부에 C함수 선언(Linked List)` 한 경우 어떻게 될까요?
3. 결국 타고, 타고 올라가다 보면 `전역 컨텍스트의 LexicalEnvironment를 참조`하게 됩니다.
4. 항상 outer는 오직 자신이 **선언된 시점**의 LexicalEnvironment를 참조하고 있으므로, **가장 가까운 요소부터 차례대로 접근 가능**
5. 결론 : 무조건 스코프 체인 상에서 가장 먼저 발견된 식별자에게만 접근이 가능

   (대부분의 경우 text는 예시보다 항상 어렵죠 ^^;;;; 예시를 통해 볼께요)

```
// 아래 코드를 여러분이 직접 call stack을 그려가며 scope 관점에서 변수에 접근해보세요!
// 어려우신 분들은 강의를 한번 더 돌려보시기를 권장드려요 :)
var a = 1;
var outer = function() {
  var inner = function() {
    console.log(a); // 이 값은 뭐가 나올지 예상해보세요! 이유는 뭐죠? scope 관점에서!
    var a = 3;
  };
  inner();
  console.log(a); // 이 값은 또 뭐가 나올까요? 이유는요? scope 관점에서!
};
outer();
console.log(a); // 이 값은 뭐가 나올까요? 마찬가지로 이유도!
```

   위의 과정을 통해 이제는 **아래의 한 문장**이 이해가 되었으면 좋겠어요 🙏🙏

   > 각각의 실행 컨텍스트는 LE 안에 **record와 outer를 가지고 있고**, outer 안에는 그 실행 컨텍스트가 **선언될 당시의 LE정보**가 다 들어있으니 **scope chain**에 의해 상위 컨텍스트의 record를 읽어올 수 있다.

3. this(정의, 활용방법, 바인딩, call, apply, bind)

이번 시간에는 this에 대해 다루게 됩니다. 다른 객체지향 언어에서의 this는 곧 **클래스로 생성한 인스턴스**를 말합니다. 그러나 자바스크립트에서는 **this가 어디에서나** 사용될 수 있어요.
this가 상황별로 어떻게 달라지는지, 이유는 뭔지, this를 추적하는 방법 등을 알아 볼 예정입니다.

▼ (1) 상황에 따라 달라지는 this

잊지 않으셨겠죠? 또 리마인드!!

> 💡 - **실행 컨텍스트**는 **실행할 코드에 제공할 환경 정보**들을 모아놓은 객체이다.
> - 그 객체 안에는 3가지가 존재한다.
> ✓ VariableEnvironment
> ✓ LexicalEnvironment
> ✓ **ThisBindings**

1. this는 실행 컨텍스트가 생성될 때 결정돼요. 이 말을 this를 bind한다(=묶는다) 라고도 하죠. 다시 말하면, `this는 함수를 호출할 때 결정된다.` 라고 할 수 있습니다.
   a. 전역 공간에서의 this
      i. 전역 공간에서 this는 **전역 객체**를 가리켜요.
      ii. 런타임 환경에 따라 this는 window(브라우저 환경) 또는 global(node 환경)를 각각 가리킵니다.

      > 💡 **런타임 환경?**
      >
      > 여러분들이 javascript로 만들어놓은 프로그램이 구동중인 환경을 말하죠. 우리는 node 파일이름.js로 vscode 상에서 구동하고 있으니 **node 환경**이라고 할 수 있구요. html 파일 안에 숨겨놓아서 크롬브라우저 등에서 연다고 한다면 **브라우저 환경**이라고 할 수 있겠네요.

      <브라우저 환경 this 확인>

      ```
      console.log(this);
      console.log(window);
      console.log(this === window);
      ```

      <node 환경 this 확인>

      ```
      console.log(this);
      console.log(global);
      console.log(this === global);
      ```

2. 메서드로서 호출할 때 그 메서드 내부에서의 this
   a. 함수 vs 메서드

      함수와 메서드, 상당히 비슷해 보이지만 엄연한 차이가 존재합니다. 기준은 `독립성`이에요. 함수는 그 자체로 독립적인 기능을 수행해요.

      ```
      함수명();
      ```

      그러나 메서드는 자신을 호출한 대상 객체에 대한 동작을 수행해요. 다음과 같이요.

```
객체.메서드명();
```

b. this의 할당

```
// CASE1 : 함수
// 호출 주체를 명시할 수 없기 때문에 this는 전역 객체를 의미해요.
var func = function (x) {
  console.log(this, x);
};
func(1); // Window { ... } 1

// CASE2 : 메서드
// 호출 주체를 명시할 수 있기 때문에 this는 해당 객체(obj)를 의미해요.
// obj는 곧 { method: f }를 의미하죠?
var obj = {
  method: func,
};
obj.method(2); // { method: f } 2
```

c. 함수로서의 호출과 메서드로서의 호출 구분 기준 : . []

아래 예시도 같아요! 점(.)으로 호출하든, 대괄호([])로 호출하든 결과는 같습니다 😉

```
var obj = {
  method: function (x) { console.log(this, x) }
};
obj.method(1); // { method: f } 1
obj['method'](2); // { method: f } 2
```

d. 메서드 내부에서의 this

위의 내용에서 보았듯, this에는 `호출을 누가 했는지에 대한 정보`가 담겨요.

```
var obj = {
  methodA: function () { console.log(this) },
  inner: {
    methodB: function() { console.log(this) },
  }
};

obj.methodA();             // this === obj
obj['methodA']();          // this === obj

obj.inner.methodB();       // this === obj.inner
obj.inner['methodB']();    // this === obj.inner
obj['inner'].methodB();    // this === obj.inner
obj['inner']['methodB'](); // this === obj.inner
```

3. 함수로서 호출할 때 그 함수 내부에서의 this

a. 함수 내부에서의 this

i. 어떤 함수를 함수로서 호출할 경우, this는 지정되지 않아요(호출 주체가 알 수 없으니까요)

ii. 실행컨텍스트를 활성화할 당시 this가 지정되지 않은 경우, this는 전역 객체를 의미하죠

iii. 따라서, 함수로서 **'독립적으로' 호출할 때**는 `this는 항상 전역객체를 가리킨다`는 것을 주의하길 바래요 👍

b. 메서드의 **내부함수**에서의 this

i. 예외는 없습니다! 메서드의 내부라고 해도, 함수로서 호출한다면 this는 전역 객체를 의미해요!

```
var obj1 = {
  outer: function() {
    console.log(this); // (1)
    var innerFunc = function() {
      console.log(this); // (2), (3)
    }
    innerFunc();

    var obj2 = {
      innerMethod: innerFunc
    };
    obj2.innerMethod();
```

```
  }
};
obj1.outer();
```

위 코드의 실행 결과 (1), (2), (3)을 예측해볼까요?

> (1) : obj1, (2) : 전역객체, (3) : obj2

맞았다면 여러분은 this에 대해 정말 많이 이해하신 거에요 👍👍

> ❗ this 바인딩에 관해서는 함수를 실행하는 당시의 주변 환경(메서드 내부인지, 함수 내부인지)은 중요하지 않고, **오직 해당 함수를 호출하는 구문 앞에 점 또는 대괄호 표기가 있는지**가 관건이라는 것을 알 수 있습니다.

c. 메서드의 내부 함수에서의 this 우회

this에 대해서 이해는 하겠지만… 사용자 입장에서. 즉, 개발자 입장에서 이게 쉽게 받아들여지시나요? 그렇지 않죠? 그렇기 때문에 **우회할 수 있는 방법**을 우리는 찾아볼 수 있습니다.

1. 변수를 활용하는 방법

   내부 스코프에 이미 존재하는 this를 별도의 변수(ex : self)에 할당하는 방법이에요!

```
var obj1 = {
  outer: function() {
    console.log(this); // (1) outer

    // AS-IS
    var innerFunc1 = function() {
      console.log(this); // (2) 전역객체
    }
    innerFunc1();

    // TO-BE
    var self = this;
    var innerFunc2 = function() {
      console.log(self); // (3) outer
    };
    innerFunc2();
  }
};

// 메서드 호출 부분
obj1.outer();
```

2. 화살표 함수(=this를 바인딩하지 않는 함수)

   a. ES6에서 처음 도입된 화살표 함수는, 실행 컨텍스트를 생성할 때 this 바인딩 과정 자체가 없습니다(따라서, this는 이전의 값-상위값-이 유지돼요 / ES6에서는 함수 내부에서 this가 전역객체를 바라보는 문제 때문에 화살표함수를 도입했어요!)

   > **일반 함수와 화살표 함수의 가장 큰 차이점은 무엇인가요? 라고 물으면 여러분은 무엇이라고 답해야 할까요?**
   > **> this binding 여부**가 가장 적절한 답입니다 😉

```
var obj = {
  outer: function() {
    console.log(this); // (1) obj
    var innerFunc = () => {
      console.log(this); // (2) obj
    };
    innerFunc();
  }
}

obj.outer();
```

4. 콜백 함수 호출 시 그 함수 내부에서의 this

   우리는 앞선 과정에서 콜백 함수를 다음과 같이 정의한 적이 있어요.

"어떠한 함수, 메서드의 인자(매개변수)로 넘겨주는 함수"

이 때, 콜백함수 내부의 this는 해당 콜백함수를 넘겨받은 함수(메서드)가 정한 규칙에 따라 값이 결정된답니다. **콜백 함수도 함수**기 때문에 this는 전역 객체를 참조하지만(호출 주체가 없잖아요), 콜백함수를 넘겨받은 함수에서 **콜백 함수에 별도로 this를 지정한 경우**는 예외적으로 그 대상을 참조하게 되어있어요. 다음 예시를 통해 구체적으로 알아봅시다.

> ❗ **로직을 이해하는 것 보다는, this의 상태를 이해하는 것이 더 중요해요!**

```
// 별도 지정 없음 : 전역객체
setTimeout(function () { console.log(this) }, 300);

// 별도 지정 없음 : 전역객체
[1, 2, 3, 4, 5].forEach(function(x) {
  console.log(this, x);
});

// addListener 안에서의 this는 항상 호출한 주체의 element를 return하도록 설계되었음
// 따라서 this는 button을 의미함
document.body.innerHTML += '<button id="a">클릭</button>';
document.body.querySelector('#a').addEventListener('click', function(e) {
  console.log(this, e);
});
```

1. setTimeout 함수, forEach 메서드는 콜백 함수를 호출할 때 대상이 될 this를 지정하지 않으므로, this는 곧 window객체
2. addEventListner 메서드는 콜백 함수 호출 시, 자신의 this를 상속하므로, this는 addEventListner의 앞부분(button 태그)

5. 생성자 함수 내부에서의 this
   a. 생성자 : 구체적인 인스턴스(어려우면 객체로 이해!)를 만들기 위한 일종의 틀
   b. 공통 속성들이 이미 준비돼 있어요.

   `객체를 생성하는 방법`에서 이미 언급한 적이 있었죠. 거기서 this를 본 적이 있었어요!

```
var Cat = function (name, age) {
  this.bark = '야옹';
  this.name = name;
  this.age = age;
};

var choco = new Cat('초코', 7); //this : choco
var nabi = new Cat('나비', 5);  //this : nabi
```

▼ (2) 명시적 this 바인딩

자동으로 부여되는 상황별 this의 규칙을 깨고 this에 별도의 값을 저장하는 방법입니다.

크게, `call / apply / bind`에 대해 알아보겠습니다.

1. call 메서드
   a. 호출 주체인 함수를 즉시 실행하는 명령어에요.
   b. call명령어를 사용하여, 첫 번째 매개변수에 this로 binding할 객체를 넣어주면 **명시적으로 binding**할 수 있어요. 쉽죠?
   c. 예시를 통해 확인해봅시다.

```
var func = function (a, b, c) {
  console.log(this, a, b, c);
};

// no binding
func(1, 2, 3); // Window{ ... } 1 2 3

// 명시적 binding
// func 안에 this에는 {x: 1}이 binding돼요
func.call({ x: 1 }, 4, 5, 6}; // { x: 1 } 4 5 6
```

아래 예시를 통해, 예상되는 this가 있음에도 일부러 바꾸는 연습을 해봅시다!

```
var obj = {
  a: 1,
  method: function (x, y) {
    console.log(this.a, x, y);
  }
};

obj.method(2, 3); // 1 2 3
obj.method.call({ a: 4 }, 5, 6); // 4 5 6
```

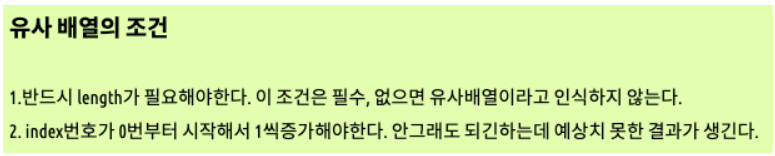
2. apply 메서드
   a. call 메서드와 완전히 동일해요! 다만, this에 binding할 객체는 똑같이 넣어주고 나머지 부분만 배열 형태로 넘겨줍니다.
   b. 예시

```
var func = function (a, b, c) {
  console.log(this, a, b, c);
};
func.apply({ x: 1 }, [4, 5, 6]); // { x: 1 } 4 5 6

var obj = {
  a: 1,
  method: function (x, y) {
    console.log(this.a, x, y);
  }
};

obj.method.apply({ a: 4 }, [5, 6]); // 4 5 6
```

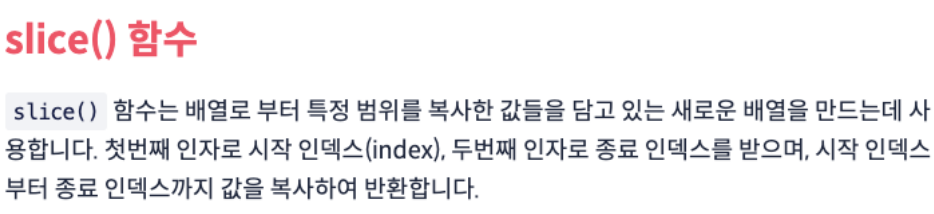
3. call / apply 메서드 활용

   물론 this binding을 위해 call, apply method를 사용하기도 하지만 더 유용한 측면도 있습니다.

   1. 유사배열객체(array-like-object)에 배열 메서드를 적용



(출처 : https://kamang-it.tistory.com/entry/JavaScript15유사배열-객체Arraylike-Objects)



(출처 : https://www.daleseo.com/js-array-slice-splice/)

```
//객체에는 배열 메서드를 직접 적용할 수 없어요.
//유사배열객체에는 call 또는 apply 메서드를 이용해 배열 메서드를 차용할 수 있어요.
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};
Array.prototype.push.call(obj, 'd');
console.log(obj); // { 0: 'a', 1: 'b', 2: 'c', 3: 'd', length: 4 }

var arr = Array.prototype.slice.call(obj);
console.log(arr); // [ 'a', 'b', 'c', 'd' ]
```

   2. Array.from 메서드(ES6)

사실, call/apply를 통해 this binding을 하는 것이 아니라 `객체 → 배열`로의 형 변환 만을 위해서도 쓸 수 있지만 원래 의도와는 거리가 먼 방법이라 할 수 있습니다.

따라서, ES6에서는 `Array.from`이라는 방법을 제시했는데요. 아주 편리해요!

```
// 유사배열
var obj = {
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
};

// 객체 -> 배열
var arr = Array.from(obj);

// 찍어보면 배열이 출력됩니다.
console.log(arr);
```

3. 생성자 내부에서 다른 생성자를 호출(공통된 내용의 반복 제거)

   `Student`, `Employee` 모두 `Person`입니다. **name**과 **gender** 속성 모두 필요하죠. 그러니 `Student`와 `Employee` 인스턴스를 만들 때 마다 세 가지 속성을 모두 각 생성자 함수에 넣기 보다는 `Person`이라는 생성자 함수를 별도로 빼는게 '구조화'에 도움이 더 되겠네요 😉

```
function Person(name, gender) {
  this.name = name;
  this.gender = gender;
}
function Student(name, gender, school) {
  Person.call(this, name, gender); // 여기서 this는 student 인스턴스!
  this.school = school;
}
function Employee(name, gender, company) {
  Person.apply(this, [name, gender]); // 여기서 this는 employee 인스턴스!
  this.company = company;
}
var kd = new Student('길동', 'male', '서울대');
var ks = new Employee('길순', 'female', '삼성');
```

4. 여러 인수를 묶어 하나의 배열로 전달할 때 apply 사용할 수 있어요.
   a. apply를 통해 비효율적인 예시를 효율적인 예시로 바꿔봅시다!

```
//비효율
var numbers = [10, 20, 3, 16, 45];
var max = min = numbers[0];
numbers.forEach(function(number) {
  // 현재 돌아가는 숫자가 max값 보다 큰 경우
  if (number > max) {
    // max 값을 교체
    max = number;
  }

  // 현재 돌아가는 숫자가 min값 보다 작은 경우
  if (number < min) {
    // min 값을 교체
    min = number;
  }
});

console.log(max, min);
```

어떤가요? 코드가 너무 길고 가독성이 떨어집니다. 😭 apply를 적용해보면 어떻게 될까요?

```
//효율
var numbers = [10, 20, 3, 16, 45];
var max = Math.max.apply(null, numbers);
var min = Math.min.apply(null, numbers);
console.log(max, min);

// 펼치기 연산자(Spread Operation)를 통하면 더 간편하게 해결도 가능해요
const numbers = [10, 20, 3, 16, 45];
const max = Math.max(...numbers);
```

```
const min = Math.min(...numbers);
console.log(max min);
```

4. bind 메서드
   a. call과 비슷해 보입니다. 하지만, 즉시 call과는 다르게 즉시 호출하지는 않고 넘겨받은 this 및 인수들을 바탕으로 새로운 함수를 반환하는 메서드라고 보시면 돼요!
   b. 목적
      i. 함수에 **this를 미리** 적용해요!
      ii. **부분 적용 함수** 구현할 때 용이합니다.
   c. 예시

```
var func = function (a, b, c, d) {
  console.log(this, a, b, c, d);
};
func(1, 2, 3, 4); // window객체

// 함수에 this 미리 적용
var bindFunc1 = func.bind({ x: 1 }); // 바로 호출되지는 않아요! 그 외에는 같아요.
bindFunc1(5, 6, 7, 8); // { x: 1 } 5 6 7 8

// 부분 적용 함수 구현
var bindFunc2 = func.bind({ x: 1 }, 4, 5); // 4와 5를 미리 적용
bindFunc2(6, 7); // { x: 1 } 4 5 6 7
bindFunc2(8, 9); // { x: 1 } 4 5 8 9
```

   d. name 프로퍼티
      i. bind 메서드를 적용해서 새로 만든 함수는 name 프로퍼티에 ‘bound’ 라는 접두어가 붙습니다(추적하기가 쉽죠!)

```
var func = function (a, b, c, d) {
  console.log(this, a, b, c, d);
};
var bindFunc = func.bind({ x:1 }, 4, 5);

// func와 bindFunc의 name 프로퍼티의 차이를 살펴보세요!
console.log(func.name); // func
console.log(bindFunc.name); // bound func
```

   e. 상위 컨텍스트의 this를 내부함수나 콜백 함수에 전달하기
      i. 내부함수
         1. 메서드의 내부함수에서 메서드의 this를 그대로 사용하기 위한 방법이에요(이전에는 내부함수에 this를 전달하기 위해 self를 썼었던 것 기억 나시나요?)

```
// TO-BE
var self = this;
var innerFunc2 = function() {
  console.log(self); // (3) outer
};
innerFunc2();
```

         2. self 등의 변수를 활용한 우회법보다 call, apply, bind를 사용하면 깔끔하게 처리 가능하기 때문에 이렇게 이용하는게 더 낫겠어요 🙂

```
var obj = {
  outer: function() {
    console.log(this); // obj
    var innerFunc = function () {
      console.log(this);
    };

    // call을 이용해서 즉시실행하면서 this를 넘겨주었습니다
    innerFunc.call(this); // obj
  }
};
obj.outer();
```

이번엔, call이 아니라 bind를 이용해볼께요!

```
var obj = {
  outer: function() {
    console.log(this);
    var innerFunc = function () {
      console.log(this);
    }.bind(this); // innerFunc에 this를 결합한 새로운 함수를 할당
    innerFunc();
  }
};
obj.outer();
```

ii. 콜백함수

1. **콜백함수도 함수**이기 때문에, 함수가 인자로 전달될 때는 함수 자체로 전달해요.

   (this가 유실되죠!)

2. bind메서드를 이용해 this를 입맛에 맞게 변경 가능합니다.

```
var obj = {
  logThis: function () {
    console.log(this);
  },
  logThisLater1: function () {
    // 0.5초를 기다렸다가 출력해요. 정상동작하지 않아요.
    // 콜백함수도 함수이기 때문에 this를 bind해주지 않아서 잃어버렸어요!(유실)
    setTimeout(this.logThis, 500);
  },
  logThisLater2: function () {
    // 1초를 기다렸다가 출력해요. 정상동작해요.
    // 콜백함수에 this를 bind 해주었기 때문이죠.
    setTimeout(this.logThis.bind(this), 1000);
  }
};

obj.logThisLater1();
obj.logThisLater2();
```

5. 화살표 함수의 예외사항

   a. 화살표 함수는 실행 컨텍스트 생성 시, this를 바인딩하는 과정이 제외된다고 했었죠!

   b. 이 함수 내부에는 this의 할당과정(바인딩 과정)이 아에 없으며, 접근코자 하면 스코프체인상 가장 가까운 this에 접근하게 됨

   c. this우회, call, apply, bind보다 편리한 방법

```
var obj = {
  outer: function () {
    console.log(this);
    var innerFunc = () => {
      console.log(this);
    };
    innerFunc();
  };
};
obj.outer();
```

4. 숙제

▼ 01. 나이든 유저

- 가장 아래의 코드가 실행 되었을 때, “Passed ~” 가 출력되도록 getAge 함수를 채워주세요

```
var user = {
    name: "john",
    age: 20,
}

var getAged = function (user, passedTime) {
    // 여기를 작성해 주세요!
```

```
}


var agedUser = getAged(user, 6);

var agedUserMustBeDifferentFromUser = function (user1, user2) {
    if (!user2) {
        console.log("Failed! user2 doesn't exist!");
    } else if (user1 !== user2) {
        console.log("Passed! If you become older, you will be different from you in the past!")
    } else {
        console.log("Failed! User same with past one");
    }
}

agedUserMustBeDifferentFromUser(user, agedUser);
```

▼ 02. 어떤 매치가 성사될까?

- 출력의 결과를 제출해주세요, 그리고 그 이유를 최대한 상세히 설명해주세요
- **주의사항 : 브라우저에서 테스트해주세요(Chrome → 개발자 도구 → 콘솔에 입력하여 실행)**

```
var fullname = 'Ciryl Gane'

var fighter = {
    fullname: 'John Jones',
    opponent: {
        fullname: 'Francis Ngannou',
        getFullname: function () {
            return this.fullname;
        }
    },

    getName: function() {
        return this.fullname;
    },

    getFirstName: () => {
        return this.fullname.split(' ')[0];
    },

    getLastName: (function() {
        return this.fullname.split(' ')[1];
    })()

}

console.log('Not', fighter.opponent.getFullname(), 'VS', fighter.getName());
console.log('It is', fighter.getName(), 'VS', fighter.getFirstName(), fighter.getLastName);
```

▼ 정답 코드 / 해설 영상

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/187944f2-bc00-441d-a25d-389d3083487e/3%E1%84%8C%E1%85%AE%E1%84%8E%E1%85%A1(1).mp4

```
var user = {
    name: "john",
    age: 20,
}

// 객체 만들어 프로퍼티 복사하기
var getAged = function (user, passedTime) {
    var result = {};
    for (var prop in user) {
        result[prop] = user[prop];
    }
    result.age += passedTime;
    return result;
}


var agedUser = getAged(user, 6);
```

```
var agedUserMustBeDifferentFromUser = function (user1, user2) {
    if (user1 !== user2) {
        console.log("Passed! If you become older, you will be different from you in the past!")
    } else {
        console.log("Failed! User same with past one");
    }
}

agedUserMustBeDifferentFromUser(user, agedUser);
```

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/680c94b5-10bf-4955-b868-e6bb120dfa10/3%E1%84%8C%E1%85%AE%E1%84%8E%E1%85%A1(2).mp4

```
var fullname = 'Ciryl Gane'

var fighter = {
    fullname: 'John Jones',
    opponent: {
        fullname: 'Francis Ngannou',
        getFullname: function () {
            // 1. 객체 this 바인딩 : 프란시스 은가누
            return this.fullname;
        }
    },

    getName: function() {
        // 2. 객체 this 바인딩 : 존 존스
        return this.fullname;
    },

    getFirstName: () => {
        // 3. 함수 this 바인딩 : 시릴
        return this.fullname.split(' ')[0];
    },

    getLastName: (function() {
        // 3. 함수 this 바인딩 : 시릴
        return this.fullname.split(' ')[1];
    })()

}

console.log('Not', fighter.opponent.getFullname(), 'VS', fighter.getName());
console.log('It is', fighter.getName(), 'VS', fighter.getFirstName(), fighter.getLastName);
```

```
Not Francis Ngannou VS John Jones
It is John Jones VS Ciryl Gane
```