

JAVASCRIPT



작성일 :2021-11-26
Hong Hyo Sang

Content

I. Javascript 기본

1. JAVASCRIPT 기본
2. 객체
3. 함수
4. 구조분해할당
5. This
6. 실행 컨텍스트
7. Argement 객체
8. Object Prototype
9. Class
10. 콜백 패턴

1-1. JavaScript

1. Javascript 기본

01. Javascript ?

- 브라우저 자체에 내장된 해석 기능을 이용한 클라이언트 기반의 객체 지향 스크립트 언어.
- 동적인 웹 페이지를 작성하기 위해서 사용 되는 언어.
- Google V8 엔진을 통한 Server Side 개발 (Node.js)

용도

- 이벤트에 반응 하는 동작 구현
- 서버에서 정보를 받거나 정보를 전달 하기 위해 사용
- HTML 요소들의 크기나 색상을 동적으로 변경
- 대화형 콘텐츠 구현
- 사용자의 입력 검증
- Serve Side Back End 개발 (Node.js)

표현식 과 문장

- 표현식 : JavaScript에서 값을 만들어내는 간단한 코드
- 문장 : 하나 이상의 표현식 모임
- 문장의 끝은 세미콜론(;)으로 종결자 표시를 한다.

식별자

- 의미 있는 단어를 사용 하여 변수와 함수명을 만든다.
- 키워드(true, case, if ..) 등으로 함수 또는 변수명을 사용 하지 말아야 한다.
- 숫자 또는 특수 문자 ("_", "\$" 제외)로 시작 하면 안됨
- 공백 문자를 포함 할 수 없음.

변수 선언

- 숫자나 문자열과 같은 단순한 데이터 형식, 객체, 복잡한 데이터 형식을 담는다.
- var, const, let 로 선언을 하면 선언과 동시에 초기화 할 수 있다.

var : 블록 범위를 무시하고 전역 변수나 함수 지역 변수로 선언

let : 블록 유효 범위를 갖는 지역 변수 (ES6) { read/write }

const : 블록 범위의 상수를 선언합니다. 상수의 값은 재할당할 수 없으며 다시 선언할 수도 없습니다. (read)

```
function fVarExample() {
    var a = 1;
    if ( true ) {
        var a = 2;
        console.log(`if 블록 내부 변수 a = ${a}`); // a = 2
    }
    console.log(`fVarExample 내부 변수 a = ${a}`); // a = 2
}

function fLetExample() {
    let a = 1;
    if ( true ) {
        let a = 2;
        console.log(`fLetExample : if 블록 내부 변수 a = ${a}`); // a = 2
    }
    console.log(`fLetExample 내부 변수 a = ${a}`); // a = 1
}
```

1-1. JavaScript

1. Javascript 기본

carName은 let으로 선언 되고 할당 되었으므로 블록 범위에서만 참조됨

```
let globalVar = "전역";
{
  let carName = "현대";
  console.log(carName); // 현대
  carName = "기아";
  console.log(carName); // 기아
  console.log(globalVar); // 전역
}

console.log(globalVar); // 전역
console.log(carName); // Uncaught ReferenceError: carName is not defined
```

var 는 함수 스코프이므로 어디에 선언이 되어도 참조 할 수 있음
-> 호이스팅 ...

```
{
  carName = "현대";
  console.log(carName);
  var carName;
}

function fFun() {
  carName = "기아";
  console.log(carName);
  var carName;
}

fFun();
console.log(carName);
```

```
> window.carName|
< '현대'
```

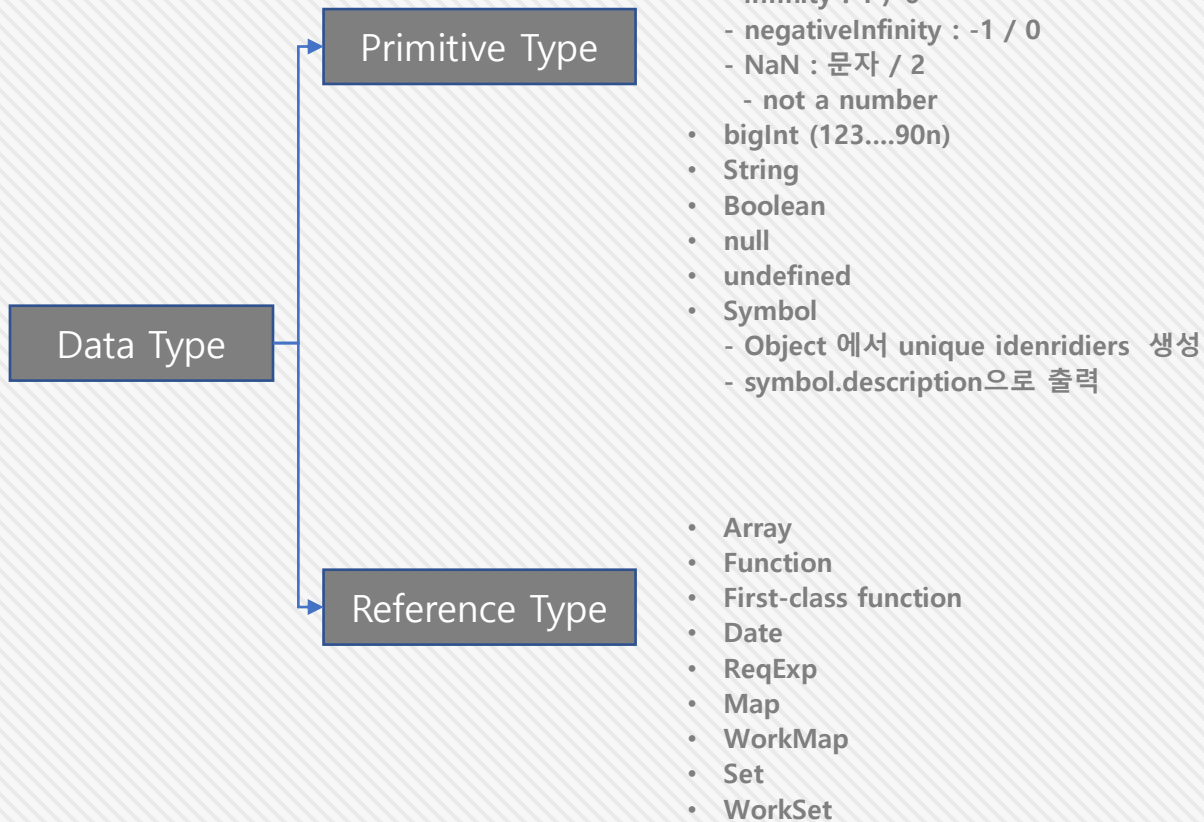
const : thread safety : 변하지 않는 값

```
const name = "현대";
const car = {
  company: "현대자동차",
  name: "소나타"
}

// name = '기아'; // Uncaught TypeError: Assignment to constant variable.
console.log(name);
console.log(car);
/*
car = {
  company: "삼성",
  name: "SM5"
}
*/ // Uncaught TypeError: Assignment to constant variable.
car.company = "삼성";
car.name = "SM5";
console.log(car);
```

1-2. JavaScript Type

데이터 형식 : 동적으로 할당 (Dynamic Typing)



- 자바 스크립트에서 변수 형식은 해당 변수가 어떤 작업을 수행 하는지를 결정 한다.
- 문자열 : 문자 데이터를 문자열로 저장, 작은 따옴표 또는 큰 따옴표로 명시
- 숫자 : 숫자를 나타내는 값을 데이터로 저장
 - $-2^{53} \sim 2^{53}$
- bigInt : 숫자(Number)의 범위를 벗어나는 값 (숫자 뒤에 n만 붙임)
- 불린 : true, false를 뜻하는 단일 비트 저장
 - false : 0, null, undefined, NaN, ""
- 배열 : 분리 구분된 일련의 데이터 요소들로 모두 하나의 변수명으로 저장
 - 0부터 시작 하는 인덱스를 이용 해서 array[index]를 사용
- 객체리터럴 : key, value로 정의 되어 있는 데이터로 object.property로 접근 할 수 있음
- null : null
- Undefined : 선언만 되고 할당이 되지 않는 것
- Template literals : `\${변수}`

```

var myString = '문자';
var myNumber = 1;
var myBoolean = true;
var myArray = ['a', 'b', 'c'];
var myObject = {"name": "홍길동", "age": "19"};
var myNull = null;

console.log(typeof myString, myString); // string 문자
console.log(typeof myNumber, myNumber); // number 1
console.log(typeof myBoolean, myBoolean); // boolean true
console.log(typeof myArray, myArray, myArray[0]);
// object [ 'a', 'b', 'c' ] a
console.log(typeof myObject, myObject, myObject["name"]);
// object { name: '홍길동', age: '19' } 홍길동
console.log(typeof myObject, myObject, myObject.name);
// object { name: '홍길동', age: '19' } 홍길동
console.log(typeof myNull, myNull); // object null
  
```

1-2. JavaScript Type

데이터 형식 : 동적으로 할당 (Dynamic Typing)

```
let variable = '문자';
console.log(variable.charAt(0))
console.log(`value : ${variable}, types: ${typeof variable}`);
variable = 10;
console.log(`value : ${variable}, types: ${typeof variable}`);
variable = variable + '20';
console.log(`value : ${variable}, types: ${typeof variable}`);
variable = variable / '20';
console.log(`value : ${variable}, types: ${typeof variable}`);
console.log(variable.charAt(0))
```

value : 문자, types: string

value : 10, types: number

value : 1020, types: string

value : 51, types: number

▶ Uncaught TypeError: variable.charAt is not a function

1-2. JavaScript Type

배열

- JavaScript에서는 배열(Array)도 객체(Object)지만 객체 순회(Iterate)를 할 때 for in을 사용해서 좋을 게 없다.
- for in은 프로토타입 체인에 있는 프로퍼티를 모두 훑는(enumerate) 데다가 객체 자신의 프로퍼티만 훑으려면 hasOwnProperty를 사용해야 하기 때문에 for보다 20배 느리다
- 배열을 만들 때 배열 생성자에 파라미터를 넣어 만드는 방법은 헛갈릴수있다. 그래서 항상 각 괄호([]) 노테이션을 이용해 배열을 만들 것을 권한다
- push, pop 보다 unshift, shift가 느리다

```
var list = [1, 2, 3, 4, 5];
for(var i = 0, l = list.length; i < l; i++) {
    console.log(list[i]); // 배열의 index 찾아옴
}

var arr = [1, 2, 3, 4, 5, 6];
arr.length = 3;
console.log(arr); // [1, 2, 3]

arr.length = 6;
arr.push(4);
console.log(arr); // [ 1, 2, 3, <3 empty items>, 4 ]
```

배열 : for, push, pop, unshift, shift

```
var lists = [1, 2, 3, 4, 5];
for(let i = 0; i < lists.length; i++) {
    console.log(lists[i]);
}

for(let list of lists) {
    console.log(list);
}

lists.forEach(function(item, index, array) {
    console.log(item, index, array);
})

lists.forEach((item, index) => console.log(item, index ));
lists.forEach((item) => console.log(item));

let arrs = [1, 2];

// 뒤에서 작업
arrs.push(3,4);
console.log(arrs); // (4) [1, 2, 3, 4]
arrs.pop();
console.log(arrs); // (2) [1, 2, 3]

// 앞에서 작업
arrs.unshift("앞1","앞2");
console.log(arrs); // (5) ['앞1', '앞2', 1, 2, 3]
arrs.shift();
console.log(arrs); // (4) ['앞2', 1, 2, 3]
```

1-2. JavaScript Type

배열 : splice, concat, indexOf, lastIndexOf, includes

```
const lists = [];
lists.push(1, 2, 3); // (3) [1, 2, 3]
console.log(lists);

// 지정한 index를 제외한 모두 삭제
lists.splice(1);
console.log(lists); // [1]

lists.push(2,3);
console.log(lists); (3) [1, 2, 3]

// 지정한 index를 삭제 하고 해당 index에 삽입
lists.splice(1, 1, 4, 5, 6);
console.log(lists); // (5) [1, 4, 5, 6, 3]

// 합치기
const addLists = [10, 11, 10];
const newList = lists.concat(addLists);
console.log(newList); // (8) [1, 4, 5, 6, 3, 10, 11, 10]

// index 찾기
console.log(newList.indexOf(10)); // 5 :: 중복이 있으면 앞에 있는 것의 index
console.log(newList.lastIndexOf(10)); // 7 :: 중복이 있으면 뒤에 있는 것의 index
console.log(newList.indexOf(2)); // -1

// 포함
console.log(newList.includes(10)); // true
console.log(newList.includes(2)); // false
```

배열 : join, split

```
const cars = ['소나타', 'SM5', '그랜저'];

// 문자열 변환
let convertString = cars.join();
console.log(convertString); // 소나타,SM5,그랜저

// 구분자 추가 하여 문자열 변환
convertString = cars.join("#");
console.log(convertString); // 소나타#SM5#그랜저

// string를 array로 변환
const carString = '소나타,SM5,그랜저';

// , 로 분리 하여 array로 변경
let convertArray = carString.split(",");
console.log(convertArray); // (3) ['소나타', 'SM5', '그랜저']

onvertArray = carString.split(", ", 2);
console.log(onvertArray); // (2) ['소나타', 'SM5']
```

배열 : reverse

```
const cars = ['소나타', 'SM5', '그랜저'];
let carsReverse = cars.reverse();
console.log(carsReverse); // (3) ['그랜저', 'SM5', '소나타']
console.log(cars); // (3) ['그랜저', 'SM5', '소나타']
```


1-2. JavaScript Type

배열 : splice와 slice 의 차이점

```
let cars = ['소나타', 'SM5', '그랜저'];
let newCars = cars.splice(0,1); // 원본이 외국됨
console.log(newCars); // ['소나타'] -> 삭제된 값
console.log(cars); // (2) ['SM5', '그랜저'] -> 남아 있는 값

cars = ['소나타', 'SM5', '그랜저'];
newCars = cars.slice(1,3); // 원본이 외국되지 않음
console.log(newCars); // (2) ['SM5', '그랜저']
newCars = cars.slice(0,1);
console.log(newCars); // ['소나타']
newCars = cars.slice(1,2);
console.log(newCars); // ['SM5']
console.log(cars); // (3) ['소나타', 'SM5', '그랜저']
```

1-3. 데이터 할당, 불변값

데이터 할당

```
var a;           // 변수 선언
a = 'abc';       // 변수 a에 데이터 할당

var b = 'def';   // 변수 선언과 데이터 할당
```

변수 영역	...	101	102	103	...
		이름: a 값: 502	이름: b 값: 503		
데이터 영역	...	501	502	503	...
			'abc'	'def'	

```
var a;           // 변수 선언
a = 'abc';       // 변수 a에 데이터 할당

a = 'xyz';       // 데이터 변경
```

변수 영역	...	101	102	103	...
		이름: a 값: 501	이름: b 값: 503		
데이터 영역	...	501	502	503	...
		'xyz'	'abc'	'def'	

불변값

- 변수(Variable)과 상수(constant)는 변경 가능성에 대한 것으로 변수 영역에 다른 데이터를 재 할당 할 수 있는 것이 변수 이다.
- 불변성은 데이터 영역의 메모리를 변경 하는 것에 대한 문제 이다.

```
var a = 'abc';

var b = 'def';
var c = 'def';

a = a + 'xyz';
b = 7;
```

재활용						변경	변경
변수 영역	...	101	102	103	
		이름: a 값: 502	이름: b 값: 502	이름: c 값: 502			
데이터 영역	...	501	502	503	504
		'abc'	'def'				

변수 영역	...	101	102	103		...
		이름: a 값: 503	이름: b 값: 504	이름: c 값: 502		
데이터 영역	...	501	502	503	504	...
		'abc'	'def'	'abcxyz'	7	

불변값 으로 GC 대상

신규 할당

1-3. 데이터 할당, 가변값

가변값

```
var obj = {
  a: 1,
  b: '문자'
};
```

변수 영역	...	101	102	103	...
		이름: obj 값: 501	이름: a 값: 502	이름: b 값: 503	
데이터 영역	...	501	502	503	...
		102, 103	1	'문자'	

- obj의 값은 메모리 501을 가리키며, 501은 102,103 변수 영역 가지고 있음

```
var obj = {
  a: 1,
  b: '문자'
};

Obj.a = 2;
```

변하지않음 변경됨

변수 영역	...	101	102	103	...	
		이름: obj 값: 501	이름: a 값: 504	이름: b 값: 503		
데이터 영역	...	501	502	503	504	
		102, 103	1 GC대상	'문자'	2	

- 기존 객체 내부 값만 바뀜

```
var arr = [3,4,5]
```

변수 영역	...	101	102	103	...	
		이름: 0 값: 501	이름: 1 값: 502	이름: 1 값: 503		
데이터 영역	...	501	502	503	504	
		3	4	5		

1-3. 데이터 할당, 가변값

가변값 - 중첩객체

```
var obj = {  
  a: 1,  
  arr: [1,2,3]  
};
```

변수 영역	...	101	102	103	104	105	106		
		이름: obj 값: 501	이름: a 값: 502	이름: arr 값: 503	이름: 0 값: 502	이름: 1 값: 504	이름: 2 값: 505		
데이터 영역	...	501	502	503	504	505			
		102, 103	1	104,105,106	2	3			

```
var obj = {  
  a: 1,  
  arr: [1,2,3]  
};  
console.log(obj);  
  
obj.arr = '문자';  
console.log(obj);
```

변하지않음		변경됨		GC대상	GC대상	GC대상			
변수 영역	...	101	102	103	104	105	106		
		이름: obj 값: 501	이름: a 값: 502	이름: arr 값: 506	이름: 0 값: 502	이름: 1 값: 504	이름: 2 값: 505		
데이터 영역	...	501	502	503	504	505	506		
		102, 103	1	104,105,106	2	3	문자		
				GC대상	GC대상	GC대상			

1-4. 데이터 할당 – 변수 비교

변수 비교

```
var a = 10;
var b = a;
var obja = { c:10, d:'문자' }
var objb = obja;
```

```
var a = 10;
var b = a;
var obja = { c:10, d:'문자' }
var objb = obja;
```

```
b = 20;
objb.c = 30;
```

```
objc = { c:20, d:'문자'};
```

```
console.log(obja === objb ? true : false);
console.log(obja === objc ? true : false);
```

변수 영역	...	101	102	103	104	105	106		
		이름: a 값: 501	이름: b 값: 501	이름: obja 값: 502	이름: c 값: 501	이름: d 값: 503	이름: objb 값: 502		
데이터 영역	...	501	502	503	504	505			
		10	104,105	문자					

변경됨

변경되지 않음

변수 영역	...	101	102	103	104	105	106		
		이름: a 값: 501	이름: b 값: 504	이름: obja 값: 502	이름: c 값: 505	이름: d 값: 503	이름: objb 값: 502		
데이터 영역	...	501	502	503	504	505			
		10	104,105	문자	20	3-			

- 참조형 데이터가 '가변값' :: 내부의 프로퍼티를 변경 할 때 이다

변경됨

변수 영역	...	101	102	103	104	105	106	107	106	108		
		이름: a 값: 501	이름: b 값: 504	이름: obja 값: 502	이름: c 값: 505	이름: d 값: 503	이름: objb 값: 502	이름: objc 값: 504	이름: c 값: 504	이름: d 값: 503		
데이터 영역	...	501	502	503	504	505	504					
		10	104,105	문자	20	3-	106,108					

형 변환

- String, Number, Boolean 함수를 사용 하여 형 변환을 한다..

```
let sStr = "2";
console.log(typeof sStr, sStr); // string 2

sStr = Number(sStr);
console.log(typeof sStr, sStr); // number 2

sStr = String(sStr);
console.log(typeof sStr, sStr); // string 2

let isBoolean = "0"; // 문자는 값이 있으면 Boolean으로 변환 시 true
console.log(typeof isBoolean, isBoolean); // string 0

isBoolean = Boolean(isBoolean);
console.log(typeof isBoolean, isBoolean); // boolean true

isBoolean = ""; // 문자는 값이 없으면 Boolean으로 변환 시 false
isBoolean = Boolean(isBoolean);
console.log(typeof isBoolean, isBoolean); // boolean false

isBoolean = 0;
console.log(typeof isBoolean, isBoolean); // number 0
isBoolean = Boolean(isBoolean);
console.log(typeof isBoolean, isBoolean); // boolean false

isBoolean = 1;
console.log(typeof isBoolean, isBoolean); // number 1
isBoolean = Boolean(isBoolean);
console.log(typeof isBoolean, isBoolean); // boolean true
```

타입 캐스팅

- 다음은 모두 true
 - new Number(10) == 10; // Number.toString()이 호출되고 다시 Number로 변환된다.
 - 10 == '10'; // 스트링은 Number로 변환된다.
 - 10 == '+10'; // 이상한 스트링
 - 10 == '010'; // 엉뚱한 스트링
 - isNaN(null) == false; // null은 NaN이 아녀서 0으로 변환된다.
- 다음은 모두 false
 - 10 == 010;
 - 10 == '-10';
- 기본 타입 생성자
 - Number나 String 같은 기본 타입들의 생성자는 new 키워드가 있을 때와 없을 때 다르게 동작한다.
 - new 키워드와 함께 Number 같은 기본 타입의 생성자를 호출하면 객체를 생성하지만 new 없이 호출하면 형 변환만 시킨다

```
new Number(10) === 10; // False, Object와 Number
Number(10) === 10; // True, Number와 Number
new Number(10) + 0 === 10; // True, 타입을 자동으로 변환해주기 때문에
```

스트링으로 변환하기 : " + 10 === '10'; // true : 숫자를 빈 스트링과 더하면 쉽게 스트링으로 변환
숫자로 변환하기 : +'10' === 10;

// true : + 연산자만 앞에 붙여주면 스트링을 쉽게 숫자로 변환

Boolean으로 변환하기 :

```
!!'foo'; // true : '!' 연산자를 두 번 사용하면 쉽게 Boolean으로 변환
!!"; // false
!!'0'; // true
!!'1'; // true
!!'-1' // true
!!{}; // true
!!true; // true
```

산술 연산자

- +, -, ++, --, *, /, %

비교(조건) 연산자

- ==(값만 같음), === (값과 형식이 모두 같음), !=, !==, >, >=, <, <=, &&, ||, !

대입 연산자

- =, +=, -=, *=, /=, %=

if, switch, while, do/while, for, for/in

```
if (조건) {  
  만족;  
} else {  
  불만족  
}
```

```
switch(수식) {  
  case 값:  
    <실행코드>;  
  case 값1:  
    <실행코드>  
    break;  
  case 값2:  
    <실행코드>  
    break;  
  default :  
    <값, 값1도 아닌 경우>  
}
```

```
While (조건) {  
  조건이 false 일 때 까지  
}
```

```
do {  
  적어도 한번 실행  
} while (조건)  
  
₩
```

```
for( 대입문; 조건문; 갱신문) {  
  ....  
}  
  
continue, break
```

```
반복 될 수 있는 모든 데이터 형식을 대상으로 ""  
var myArray = ['a','b','c'];  
for (var idx in myArray) {  
  console.log(myArray[idx]);  
}
```

조건 ? 만족값 : 불만족값

객체 비교 하기

- 이중 등호 연산자 (==) 는 다음과 같은 결과가 나므로 객체 비교 사용시 주의

```
"" == "0"           // false  
0 == ""             // true  
0 == "0"            // true  
false == "false"    // false  
false == "0"        // true  
false == undefined  // false  
false == null        // false  
null == undefined   // true  
" WtWrWn" == 0       // true
```

- 삼중 등호 연산자 (===) 는 삼중 등호는 강제로 타입을 변환하지 않는다는 사실을 제외 하면 이중 등호와 동일하다
-> 삼중 등호를 사용하면 코드를 좀 더 튼튼하게 만들 수 있고, 비교하는 두 객체의 타입이 다르면 더 좋은 성능을 얻을 수도 있다

```
"" === "0"           // false  
0 === ""             // false  
0 === "0"            // false  
false === "false"    // false  
false === "0"        // false  
false === undefined  // false  
false === null        // false  
null === undefined   // false  
" WtWrWn" === 0       // false
```

- typeof 연산자, instanceof 연산자, Object.prototype.toString

1-7. 펼침연산자

...

- Object에 있는 항목을 목록으로 변환해주는 연산자로 (...)로 표시 하며 단독으로 사용 할 수 있음

```
const arr = [1,2,3];  
const sp = [...arr]; var rn = arr === sp ? true : false ; // false
```

펼침연산자를 통한 데이터 관리

```
const arr1 = [1, 2, 3];  
const arr2 = [4, 5, 6];  
  
// concat()  
const concatArr = [...arr1, ...arr2]; // [ 1, 2, 3, 4, 5, 6 ]  
  
// push()  
const pushArr = [...arr1, 7]; // [ 1, 2, 3, 7 ]  
  
// splice()  
const spliceArr = [...arr1.slice(0, 2)]; //[ 1, 2 ]
```

펼침 연산자를 이용한 데이터 분리 (구조분해 할당)

```
const obj = { name: '홍길동', post: '123-456', address: '서울 송파' };  
  
const {name, ...address} = obj;  
// obj :: { name: '홍길동', post: '123-456', address: '서울 송파' }  
// name :: 홍길동  
// address :: { post: '123-456', address: '서울 송파' }
```


단락평가

- 단락 평가란 || 연산자를 이용 해서 가장 적합한 정보를 먼저 위치 하여 정보를 확인 하는 것을 의미 한다.
- 이때 typeError가 발생 하지 않도록 주의 하여야 한다.

```
const isShotCircuiting = function(param) {  
    return param || '기본값';  
}  
  
isShotCircuiting(''); // 기본값 ( param : false )  
isShotCircuiting(null); // 기본값 ( param : false )  
isShotCircuiting(undefined); // 기본값 ( param : false )  
isShotCircuiting('지정'); // 지정 ( param : true )  
isShotCircuiting(true); // true ( param : true )  
isShotCircuiting({}); // {} ( param : true )  
isShotCircuiting([]); // [] ( param : true )  
isShotCircuiting(new Array()); // [] ( param : true )
```

객체를 사용 하는 경우 선언만 하고 항목을 지정 하지 않으면 undefined로 false로 판단 한다

```
const obj = {};  
const isShotCircuitingObj = function(obj) {  
    return obj.name || '기본값';  
};  
  
isShotCircuitingObj(obj); // 기본값 ( obj.name -> undefined )  
  
const objArray = [];  
const isShotCircuitingArray = function(obj) {  
    return obj[0] || '기본값';  
};  
isShotCircuitingArray(objArray); // 기본값 ( obj.[0] -> undefined )  
isShotCircuitingArray(obj); // 기본값 ( obj.[0] -> undefined )
```

배열처리

- .map() : 형태를 바꿀 수 있지만 길이는 유지
- .sort() : 형태나 길이는 변경 되지 않고 순서만 바뀜
- .filter() : 길이를 변경 하지만 형태는 바뀌지 않음
- .find() : 배열을 반환 하지 않음, 한 개의 테이터가 반환되고 형태는 바꾸지 않음
- .forEach() : 형태 이용, 변환 없음

```
const arrayObj = ['1.0', '기능', '1.25'];

var convertNumber = function(obj) {
  const arrayObjTem = [];
  for(let i = 0 ; i < obj.length ; i++ ) {
    const itemValue = parseFloat(obj[i]);
    if(itemValue) { // 문자는 NaN으로 해석이 되어서 false 임
      arrayObjTem.push(itemValue);
    }
  }
  return arrayObjTem;
}

console.log(convertNumber(arrayObj)); // [1, 1.25]
```

map, filter을 사용 하여 변환 한다

```
arrayObj.map(item => parseFloat(item)) // [1, NaN, 1.25]
arrayObj.map(item => parseFloat(item)).filter(item=>item) // [1, 1.25]
```

map 변환 과정

```
const addressArr = [
  { name:'김길자', address:'서울'},
  { name:'홍길동', address:'강원도'},
  { name:'김영이', address:'충청도'},
  { name:'바독이', address:'서울'},
  { name:'뱀', address:'경상도'}
];

var getAddress = function(arr) {
  const arrAddress = [];
  for(let i = 0 ; i < arr.length ; i++ ) {
    const address = arr[i].address;
    arrAddress.push(address);
  }
  return arrAddress;
}

console.log(getAddress(addressArr));
// [ '서울', '강원도', '충청도', '서울', '경상도' ]
```

1-9. 배열처리

1. Javascript 기본

map변환과정

```
const addressArr = [
  { name: '김길자', address: '서울' },
  { name: '홍길동', address: '강원도' },
  { name: '김영이', address: '충청도' },
  { name: '바독이', address: '서울' },
  { name: '뱀', address: '경상도' }
];
var getAddress = function(arr) {
  const arrAddress = [];
  for(let i = 0 ; i < arr.length ; i++ ) {
    const address = arr[i].address;
    arrAddress.push(address);
  }
  return arrAddress;
}
console.log(getAddress(addressArr));
// [ '서울', '강원도', '충청도', '서울', '경상도' ]
```

1차 함수로 변경 한다

```
var getAddress = function(arr) {
  const arrAddress = [];
  for(let i = 0 ; i < arr.length ; i++ ) {
    arrAddress.push(getAddressItem(arr[i]));
  }
  return arrAddress;
}

// 함수화
var getAddressItem = function(arrItem) {
  return arrItem.address;
}

console.log(getAddress(addressArr));
// [ '서울', '강원도', '충청도', '서울', '경상도' ]
```

map로 변경 한다

```
console.log(addressArr.map(item => getAddressItem(item)));
// [ '서울', '강원도', '충청도', '서울', '경상도' ]

// addressArr 객체안의 객체는 하나를 의미 하므로 파라미터 생략 가능
console.log(addressArr.map(getAddressItem));
// [ '서울', '강원도', '충청도', '서울', '경상도' ]
```

1-9. 배열처리

reduce(): 길이와 형태 변경 됨

```
/*
accumulator  currentValue  currentIndex  sum    array
    0           0           0         0    [0, 1, 2, 3, 4]
    0           1           1         1    [0, 1, 2, 3, 4]
    1           2           2         3    [0, 1, 2, 3, 4]
    3           3           3         6    [0, 1, 2, 3, 4]
    6           4           4        10    [0, 1, 2, 3, 4]
*/
var sum = [0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
    return accumulator + currentValue;
});
console.log(sum); // 10

/*
accumulator  currentValue  currentIndex  sum    array
    10          0           0         10    [0, 1, 2, 3, 4]
    10          1           1        11    [0, 1, 2, 3, 4]
    11          2           2        13    [0, 1, 2, 3, 4]
    13          3           3        16    [0, 1, 2, 3, 4]
    16          4           4        20    [0, 1, 2, 3, 4]
*/
// 두번째 파라미터는 초기 값을 의미함
var sumInit = [0, 1, 2, 3, 4].reduce(function(accumulator, currentValue, currentIndex, array) {
    return accumulator + currentValue;
}, 10);

console.log(sumInit); // 20
```

```
const fCallback = function(collectionValues, item) {
    return [ ...collectionValues, item];
};

const saying = ['abc', 'xyz', 'opq'];
const initialValue = ['zzz'];
const copy = saying.reduce(fCallback, initialValue);

console.log(copy);
```

1-9. 필터 처리

객체에서 특정 정보를 추출하는 방법에는 여러가지가 존재 하지만 여기서는 map, reduce를 통해서 알아 본다.

map은 객체의 항목을 순환 하면서 처리를 하고 reduce는 객체 항목을 순환 하면서 처리를 하지만 결과는 새로 생성된 객체를 통해서 만들어 지며(추가) 된다

```
const cars = [
  {name:'그랜저', company:'현대'},
  {name:'SM', company:'삼성'},
  {name:'포니', company:'현대'},
  {name:'코란도', company:'쌍용'}
];

// 현대 소속 자동차 이름
var fSearchMapCars = function(searchCompany) {
  return cars.map((car) => {
    if (car.company == searchCompany) {
      return car;
    }
  });
  // searchCompany 와 같지 않으면 undefined 이므로 필터를 사용 해서 객체만 얻음
  }).filter(car => car);
};

console.log(fSearchMapCars('현대'));
// [ { name: '그랜저', company: '현대' }, { name: '포니', company: '현대' } ]

var fSearchReduceCars = function(searchKey) {
  // targetCars의 초기값은 [] 이고 조건에 만족 하는 경우 추가 되면
  // 항목별 순환 할 때 파라미터로 전달 된다. ( 합쳐진 값 )
  // car는 cars의 현재 항목 임
  return cars.reduce((targetCars, car) => {
    if (car.company != searchKey) {
      return targetCars;
    }
    return [...targetCars, car];
  }, []);
};

console.log(fSearchReduceCars('삼성'));
// [ { name: 'SM', company: '삼성' } ]
```

```
// 아래와 같이 변경 하여도 동일 하다.
var fSearchReduceCars = function(searchKey) {
  return cars.reduce((targetCars, car) => {
    return car.company != searchKey ? targetCars : [...targetCars, car];
  }, []);
}
```

** 제조사 Count 계산

```
var fSearchReduceCarsCnt = function() {
  return cars.reduce((targetCars, car) => {
    const count = targetCars[car.company] || 0;
    const val = {
      ...targetCars,
      [car.company] : count+1
    }
    console.log("===== val :: " , val );
    return val;
  }, {});
};

console.log(fSearchReduceCarsCnt()); // { '현대': 2, '삼성': 1, '쌍용': 1 }

// 중간 로그
===== val :: { '현대': 1 }
===== val :: { '현대': 1, '삼성': 1 }
===== val :: { '현대': 2, '삼성': 1 }
===== val :: { '현대': 2, '삼성': 1, '쌍용': 1 }
```

1-10. for문

1. Javascript 기본

for 문은 객체를 순차적으로 순회 하면서 각 항목에 대한 처리를 하고 자 할 때 사용 하는 것이다.
for 문의 유형은 for, forEach, for of, for in이 있다.
for : 순차적으로 순회
forEach : 반복문이 아니라 '함수'로 인자로 함수를 받아 각 배열의 요소에 해당 함수를 적용한다.
for of : 이터러블한 객체의 순회를 도와주는 반복문
for in : Object에 있는 key에 차례로 접근하는 데 사용되는 반복문

```
const addressArr = [
  { name: '김길자', address: '서울' },
  { name: '홍길동', address: '강원도' },
  { name: '김영이', address: '충청도' },
  { name: '바독이', address: '서울' },
  { name: '뱀', address: '경상도' }
];

var getAddressFor = function(arr) {
  const arrAddress = [];
  for(let i = 0 ; i < arr.length ; i++ ) {
    if (arr[i].address.includes('서울')) {
      arrAddress.push(arr[i]);
    }
  }
  return arrAddress;
};

var getAddressForEach = function(arr) {
  const arrAddress = [];
  arr.forEach(item => {
    if (item.address.includes('서울')) {
      arrAddress.push(item);
    }
  });
  return arrAddress;
};
```

```
var getAddressForOf = function(arr) {
  const arrAddress = [];
  for ( const item of arr) {
    if (item.address.includes('서울')) {
      arrAddress.push(item);
    }
  }
  return arrAddress;
};

var getAddressForIn = function(arr) {
  const arrAddress = [];
  for ( const item in arr) {
    if (arr[item].address.includes('서울')) {
      arrAddress.push(arr[item]);
    }
  }
  return arrAddress;
};

console.log(getAddressFor(addressArr));
// [ { name: '김길자', address: '서울' }, { name: '바독이', address: '서울' } ]
console.log(getAddressForEach(addressArr));
// [ { name: '김길자', address: '서울' }, { name: '바독이', address: '서울' } ]
console.log(getAddressForOf(addressArr));
// [ { name: '김길자', address: '서울' }, { name: '바독이', address: '서울' } ]
console.log(getAddressForIn(addressArr));
// [ { name: '김길자', address: '서울' }, { name: '바독이', address: '서울' } ]
```

1-11. 배열 정리

1. Javascript 기본

배열 : find, Filter, map, some, every, reduce

```
const cars = [
  { company: '현대자동차', carname: '소나타', reservation: true, price: 10000 },
  { company: '현대자동차', carname: '그랜저', reservation: true, price: 20000 },
  { company: '삼성자동차', carname: 'SM5', reservation: false, price: 10000 },
  { company: '기아자동차', carname: '쏘렌토', reservation: true, price: 15000 },
];

// find :: 첫번째 찾은 요소를 리턴
let result = cars.find( (car) => car.price === 10000 );
console.log(result);
// => {company: '현대자동차', carname: '소나타', reservation: true, price: 10000}

// filter :: 조건에 맞는 값 리턴
result = cars.filter( (car) => car.reservation );
console.log(result);
// => 0: {company: '현대자동차', carname: '소나타', reservation: true, price: 10000}
//      1: {company: '현대자동차', carname: '그랜저', reservation: true, price: 20000}
//      2: {company: '기아자동차', carname: '쏘렌토', reservation: true, price: 15000}

// map :: 특정 조건이 맞는 것을 찾아서 배열로 리턴
let companys = cars.map( (car) => car.company );
console.log(companys); // ['현대자동차', '현대자동차', '삼성자동차', '기아자동차']
result = companys.filter((element, index)=>{
  return companys.indexOf(element) === index;
});
console.log(result);
```

```
// some :: 배열의 요소 하나라도 만족 하면 true
result = cars.some((car) => car.price <= 10000); // true
console.log(result);

// every :: 배열의 모든 요소가 만족 할 때
result = cars.every((car) => car.price <= 10000); // false
console.log(result);
console.clear();

// reduce :: 배열 요소 값을 누적
result = cars.reduce((prev, curr) => {
  console.log(prev, ', ', curr);
  console.log('-----');
  return prev + curr.price;
}, 0); // 55000

result = cars.reduce((prev, curr) => prev + curr.price, 0);
console.log(result); // 55000

// 복합
result = cars
  .map((car)=>car.price)
  .filter( (item) => item <= 10000 )
  .reduce( (prev, curr) => prev + curr, 0);
console.log(result); // 20000

result = cars
  .map((car)=>car.price)
  .sort((a,b) => a-b) // b-a 큰것 순
  .join();
console.log(result); // 10000,10000,15000,20000
```

2-1. 객체

2. 객체

01. 객체

- JavaScript는 null과 undefined를 제외한 모든 것들은 객체처럼 동작
- name/value 쌍으로 된 Property로 구성 되기 때문에 Map 처럼 사용 될 수 있음
- {}로써 이용 해서 생성을 하며 속성 값에 접근 할 때 키로 접근

Property와 Method

- Property : 객체 내부에 있는 값으로 JavaScript에서 제공 하는 모든 자료형을 가질 수 있음
- Method : 함수 자료형 속성으로 자기 자신의 속성이라는 것을 표시 할 때는 this keyword로 표현

For in 반복문

- 객체의 속성을 사용 하기 위해서 for in 반복문을 사용

모두 객체 이다

```
console.log(false.toString()); // 'false'
console.log([1,2,3,4].toString()); // '1,2,3,4'
function fun() {}
fun.count = 1;
console.log(fun.count); // 1
```

객체 생성

- const 변수명 = new 함수();
- 빈 객체 생성 : var fun = {};
- Property 가 있는 객체 생성 : const fun = { key: '함수명' } :: 객체 리터럴

접근 방법

- 직접접근 : fun.name;
- 문자로접근 : fun['name'];
- 변수로 접근
var get = 'name';
fun[get]
모두 '함수명'을 의미함

property 삭제

```
• delete 사용
var nameObj = { first:1, middle:2, last:3};
nameObj.first = '홍';
nameObj.middle = '길동'
delete nameObj.last;
for ( var i in name ) {
  if ( name.hasOwnProperty(i) {
    console.log(i, " + name[i];
  }
}
```

```
function Add(a, b) {
  this.a = a;
  this.b = b;
  this.result = ()=>{
    return this.a + this.b;
  }
}

const fAdd = new Add(2,3);
console.log("fAdd :: ", fAdd);
// fAdd :: Add { a: 2, b: 3, result: [Function (anonymous)] }
console.log("fAdd :: fAdd.result :: ", fAdd.result());
// fAdd :: fAdd.result :: 5
```


2-2. 객체 리터럴

객체 리터럴

- {} 로 감싸고 key:object를 쉼표(,)로 구분
- 생성한 객체는 언제라도 변경 가능 하며, 내장 네이티브 객체의 프로퍼티도 대부분 변경이 가능 하다.

```
const person = {};
person.name = '홍길동';
person.getName = function() {
    return person.name;
};
console.log("person :: ", person.getName()); // person :: 홍길동
// 함수 재 정의
person.getName = function() {
    return "홍당무";
};
console.log("person :: ", person.getName()); // person :: 홍당무
```

```
var person = {
    name : '홍길동',
    getName : function () {
        return this.name;
    }
};
console.log("person ::", person.getName()); // person :: 홍길동
person.name = "홍당무";
console.log("person ::", person.getName()); // person :: 홍당무
```

객체 생성시 new 연산자를 사용 하지 마라

-> 생성자에 파라미터로 전달 시 해당 값의 형으로 결정 되므로

```
var person = new Object();
console.log(person.constructor === Object); // true
var person = new Object(1);
console.log(person.constructor === Object); // false
console.log(person.constructor === Number); // true
```

배열 리터럴

```
console.log(typeof a); // 배열도 객체 이므로 object
console.log(a.constructor); // [Function: Array]
```

```
var b = ["1","2","3"];
console.log(typeof b); // 배열도 객체 이므로 object
console.log(b.constructor); // [Function: Array]
```

```
var aa = new Array(3);
console.log(typeof aa); // object
console.log(aa.constructor); // [Function: Array]
console.log(aa.length); // 3
console.log(aa[0]); // undefined
```

```
var bb = [3];
console.log(bb.length); // 1
console.log(bb[0]); // 3
```

2-3. 불변객체

불변 객체 만드는 방법

- 전달 받은 객체가 변경이 되어도 원본 객체는 변하지 말아야 할 때 불변 객체를 만든다.

```
var obj = {
  a: 1,
  b: '문자'
}

var fChgObj = function(obj, b) {
  var obj2 = obj;
  obj2.b = b;
  return obj2
}

var obj3 = fChgObj(obj, '숫자');
console.log(obj === obj3)
```

```
var obj = {
  a: 1,
  b: '문자'
}

var fChgObj = function(obj, b) {
  return {
    a: obj.a,
    b: b
  }
}

var obj3 = fChgObj(obj, '숫자');
console.log(obj === obj3);
```

```
var obj = {
  a: 1,
  b: '문자'
}

// 얕은 복사(shallow copy) : 주소 값 복사
var copyObject = function(sourceObj) {
  var result = {};
  for( var prop in sourceObj) {
    result[prop] = sourceObj[prop];
  }
  return result;
}

var fChgObj = copyObject(obj);
fChgObj.b = '숫자';

console.log(obj === fChgObj);
```

2-4. 객체 카피

2. 객체

불변 객체 만드는 방법 - 얕은 카피, 깊은 카피,

```
var obj = {
  a: 1,
  b: { a:1, b: '문자' }
}

var shallowCopy = function(target) {
  let copy = {};
  for( var prop in target) {
    copy[prop] = target[prop];
  }
  return copy;
}

var dobj = shallowCopy(obj);
dobj.a = 2;
obj.b.a = 2;
dobj.b.b = '숫자';

console.log(obj);           // { a: 1, b: { a: 2, b: '숫자' } }
console.log(dobj);          // { a: 2, b: { a: 2, b: '숫자' } }
console.log(obj.a === dobj.a );    // false
console.log(obj.b.a === dobj.b.a ); // true
console.log(obj.b.b === dobj.b.b ); // true
```

```
var obj = {
  a: 1,
  b: { a:1, b: '문자' }
}

var deepCopy = function (target) {
  let copy = {};

  if ( typeof target === 'object' && target !== null ) {
    for (let prop in target) {
      copy[prop] = deepCopy(target[prop]);
    }
  } else {
    copy = target;
  }

  return copy;
}

var dobj = deepCopy(obj);
dobj.a = 2;
obj.b.a = 2;
dobj.b.b = '숫자';

console.log(obj);           // { a: 1, b: { a: 2, b: '문자' } }
console.log(dobj);          // { a: 2, b: { a: 1, b: '숫자' } }
console.log(obj.a === dobj.a );    // false
console.log(obj.b.a === dobj.b.a ); // false
console.log(obj.b.b === dobj.b.b ); // false
```

3-1. 함수

01. 함수

- JavaScript는 함수 자체가 다른 함수의 인자가 될 수 있다. 즉 First Class Object
- 함수는 변수는 Scope를 결정 하고 private변수 또는 메소드 뿐만 아니라 함수의 특징을 이용 하여 public 속성과 메소드를 제공 하며 자바 스크립트 모듈을 작성 한다.
- 함수는 하나의 기능을 제공해야 한다 -> 단일책임

First Class Object

- 변수에 저장 할 수 있어야 한다.
- 함수를 파라미터로 전달할 수 있어야 한다.
- 함수의 반환값으로 사용할 수 있어야 한다.
- 자료 구조에 저장할 수 있어야 한다.



함수와 익명 함수

- 함수를 정의 하는 방법.
 - Function 객체를 사용 하는 방법
 - 연산자인 Function을 사용 하는 방법
 - > 함수 선언문 (Function Declaration)
 - > 함수 표현식 (Function Expression)

- 함수 선언식으로 표현 하면 로딩 시점에 VO(Variable Object)에 적재 하므로 어느 곳에서도 접근 가능.
- 함수 표현식으로 표현 하면 Runtime시점에 적재
 - > 규모가 큰 프로젝트 인 경우 고려 사항

함수 선언 방법	예제
함수 선언식 (Function Declaration)	function fun() { ... }
기명 함수 표현식 (Named Function Expression)	var fun = function fun() { ... }
익명 함수 표현식 (Anonymous Function Expression) -> 함수 리터럴	var fun = function() { ... }
기명 즉시실행함수(named immediately-invoked function expression)	(function fun() { ... } ());
익명 즉시실행함수(immediately-invoked function expression)	(function() { ... })();

3-1. 함수

3. 함수

함수 표현식에 의한 함수 호출 과 즉시 실행 함수 호출

```
var func = function() {  
    console.log("함수 표현식에 의한 명시적인 호출");  
};
```

```
func(); // 함수 표현식에 의한 명시적인 호출
```

```
(function() {  
    console.log("즉시 실행 함수");  
})(); // 즉시 실행 함수
```

변수에 즉시 실행 함수를 할당 후 실행

```
var app = (function() {  
    var name = "홍길동";  
    return {  
        rName : name  
    };  
})();  
  
console.log(app().rName); // 홍길동
```

기명 함수 사용

```
var fPrintName= function(name) {  
    console.log("나의 이름은 " + name + "이다");  
};  
  
fPrintName("홍길동"); // 나의 이름은 홍길동이다  
  
(function(name) {  
    console.log("즉시 실행 :: 나의 이름은 " + name + "이다");  
})("홍길동"); // 즉시 실행 :: 나의 이름은 홍길동이다
```

익명 함수이용 한 scope영역 관리

```
var fFun= {  
    fFun01: function() {  
        let name = "홍";  
        return name;  
    },  
    fFun02: function(){  
        let name = "당무";  
        return name;  
    },  
    fFun03: function(){  
        return this.fFun01() + this.fFun02();  
    }  
}  
  
console.log(fFun.fFun01()); // 홍  
console.log(fFun.fFun02()); // 당무  
console.log(fFun.fFun03()); // 홍당무
```

default parameters (ES6)

```
function fPrint(mag, actionType = '홈') {  
    console.log(`메세지 : ${mag}, 출처:${actionType}`);  
}  
  
fPrint('안녕');  
fPrint('안녕', '회사');
```

메세지 : 안녕, 출처:홈

메세지 : 안녕, 출처:회사

rest parameters (ES6) : 파라미터를 펼침연산자로 넘김

```
function fPrint(...mag) {  
    console.log(`메세지 : ${mag}`);  
}  
  
fPrint('안녕', '회사'); // 메세지 : 안녕,회사
```

3-2. 함수-생성자

생성자

- new 키워드로 호출되는 함수가 생성자
- 생성자로 호출된 함수의 this 객체는 새로 생성된 객체를 가리키고, 새로 만든 객체의 prototype 에는 생성자의 prototype이 할당된다
- 전역범위는 전체를 의미 하며 지역 범위는 정의 된 함수 내에서만 참조 되는 것을 의미 함
- 생성자에 명시적인 return 구문이 없으면 this가 가리키는 객체를 반환한다

```
function Person(name) {
  this.name = name;
}
Person.prototype.getName = function() {
  console.log(this.name);
};

const fPerson = new Person("홍당무");
fPerson.getName(); // 홍당무
```

- 생성자에 명시적인 return 문이 있는 경우에는 반환하는 값이 객체인 경우에만 그 값을 반환한다.

```
function car() {
  return '현대';
}
console.log(car()); // 현대
console.log(new car()); // 새로운 객체를 반환 --> car {}

function person() {
  this.someValue = 2;
  return {
    name: '한국인'
  };
}
console.log(person()); // { name: '한국인' }
console.log(new person()); // { name: '한국인' }
```

- new 키워드가 없으면 그 함수는 객체를 반환하지 않는다.

```
function Fun() {
  this.hasEyePatch = true; // 전역 객체를 준비!
  firstName = "길동";
  var var1 = "var1"
}
var fFun = Fun();
console.log(fFun); // undefined
console.log(hasEyePatch); // true
console.log(firstName); // 길동
console.log(fFun.firstName);
// Cannot read property 'firstName' of undefined
```

3-3. 함수-화살표 함수

화살표 함수

- Function 선언 대신 => 기호로 함수 선언
- 변수에 할당 하여 재 사용 할 수 있음
- 함수 내부에 return 문만 있는 경우 생략 가능

```
function fAdd01(x,y) {
    return x + y;
}

var fAdd02 = (x,y) => {
    return x + y;
}

var fAdd03 = (x,y) => x + y;

var fAdd04 = (x,y) => (x + y);

console.log("fAdd01 :: ", typeof fAdd01 , " 결과 :: ", fAdd01(1,2));
console.log("fAdd02 :: ", typeof fAdd02 , " 결과 :: ", fAdd02(1,2));
console.log("fAdd03 :: ", typeof fAdd03 , " 결과 :: ", fAdd03(1,2));
console.log("fAdd04 :: ", typeof fAdd04 , " 결과 :: ", fAdd04(1,2));
```

결과

```
fAdd01 :: function   결과 :: 3
fAdd02 :: function   결과 :: 3
fAdd03 :: function   결과 :: 3
fAdd04 :: function   결과 :: 3
```

- Function과 차이점 : this 바인딩

```
var fFunc = {
    name: "현대자동차",
    cars: ["아반떼", "쏘나타", "그랜저"],
    printCars: function() {
        var that = this;
        this.cars.forEach(function(car) {
            console.log(that.name, car);
        });
    }
};

fFunc.printCars();
```

현대자동차 아반떼
현대자동차 쏘나타
현대자동차 그랜저

- 상위 스코프의 this를 그대로 받는다.

```
var fFunc = {
    name: "현대자동차",
    cars: ["아반떼", "쏘나타", "그랜저"],
    printCars: function() {
        this.cars.forEach(car => {
            console.log(this.name, car);
        });
    }
};

fFunc.printCars();
```

현대자동차 아반떼
현대자동차 쏘나타
현대자동차 그랜저

3-3. 화살표 함수

01. 화살표 함수

- 함수 선언, 괄호, return문, 중괄호 등 불필요한 정보를 제거 하여 간결한 코드를 만들 수 있음

```
const car = {
  name: '소나타',
  compony: '현대자동차',
  post: '130-111',
  address: '서울시 용산구',
  detailAddress: '영업부'
};

const getFullAddress = function(post, address, detailAddress) {
  return `${post} ${address} ${detailAddress}`;
};

console.log(getFullAddress(car.post, car.address, car.detailAddress));
// 130-111 서울시 용산구 영업부

// 화살표 함수를 이용한 처리
const getFullAddressArrow = ({post, address, detailAddress})
  => `${post} ${address} ${detailAddress}`;
console.log(getFullAddressArrow(car)); // 130-111 서울시 용산구 영업부

// 화살표 함수를 이용한 처리 : 객체 리턴
const getFullAddressArrowObject = ({post, address, detailAddress})
  => ({ 주소 : `${post} ${address} ${detailAddress}` });
console.log(getFullAddressArrowObject(car));
// { '주소': '130-111 서울시 용산구 영업부' }
```

```
// 고차 함수를 이용한 우편 번호 변경 후 주소 조회
const setPost = post => {
  return ({address, detailAddress}) => {
    return `${post} ${address} ${detailAddress}`;
  };
};

const getAddress = setPost('123-567');
console.log(getAddress(car));

// setPost을 다음과 같이 변경 하여도 된다.
const setPost = post => ({address, detailAddress}) => `${post}
${address} ${detailAddress}`;
```


3-4. 테스트 가능한 함수

01. 테스트 가능한 함수 유연한 함수

- 리팩토링 가능
- 이해하기 쉽다
- 일반적으로 명확하고 버그가 적다

```
function fun01(intA) {
  // 업무 로직 .....
  let val = ...;
  return val + (intA * 10);
}

1) function fun02(p1, p2) {
  let val = fun01(p1);
  return p2 + val;
}

2) function fun02(p1, p2, fun01) {
  let val = fun01(p1);
  return p2 + val;
}
```

- 1)번 함수 :
- fun01 함수를 직접 실행, 강력한 결합
 - 테스트 코드 작성 시 fun01 함수를 고려 해서 작성 => 코드 작성 어려움
- 2)번 함수 :
- 파라미터를 받아서 실행, 느슨한 결합
 - 테스트 코드 작성 시 fun01 함수를 만들어서 작성 => 코드 작성 쉬움

* fun01 함수는 내부적으로 서버호출 할 수도 있고, 복잡한 계산 식이 있을 수도 있어서 fun02 함수 테스트시는 fun01의 최종 결과를 기준으로 작성 할 수 있음

```
// fun01함수 대신 결과만 리턴 하는 Mock 함수를 만들어서 검증
function fTest01() {
  let fun01 = function (intA) {
    return 20;
  };

  let rn = fun02(1, 2, fun01);
  console.log(rn);
}

process01();
```

```
// fun01함수 대신 함수를 주입 하여 검증
const val = fun02(10, 2, function(intA) {
  return 20;
});
console.log(val);
```

```
// fun01함수 대신 함수를 주입 하여 검증 ( 화살표 함수 사용 )
val = fun02(30, 2, (intA)=> 20);
console.log(val);
```

3-5. 고차 함수

01. 고차 함수

- 다른 함수를 반환 하는 함수
- 파라미터 단일 책임의 원칙 적용 할 수 있음

```
const car01 = {
  name: '소나타',
  compony: '현대자동차'
};

const car02 = {
  name: '그랜저',
  compony: '현대자동차'
};

const address = {
  post: '130-111',
  address: '서울시 용산구',
  detailAddress: '영업부'
};
```

```
function setCarAddress(pAddress, car) {
  let { post, address, detailAddress } = pAddress;
  let rnObj = {
    post,
    address: address + detailAddress
  };
  return {...rnObj, ...car};
}

console.log(setCarAddress(address, car01));
console.log(setCarAddress(address, car02));
```

```
{
  post: '130-111',
  address: '서울시 용산구영업부',
  name: '소나타',
  compony: '현대자동차'
}
{
  post: '130-111',
  address: '서울시 용산구영업부',
  name: '그랜저',
  compony: '현대자동차'
}
```

고차 함수를 이용해서 주소와 차이름을 분리 하여 단일 책임의 원칙 적용

```
function setCarAddress01(pAddress) {
  let { post, address, detailAddress } = pAddress;
  let rnObj = {
    post,
    address: address + detailAddress
  };
  return (car) => { return {...rnObj, ...car} };
}

console.log(setCarAddress01(address)(car01));
console.log(setCarAddress01(address)(car02));
```

```
{
  post: '130-111',
  address: '서울시 용산구영업부',
  name: '소나타',
  compony: '현대자동차'
}
{
  post: '130-111',
  address: '서울시 용산구영업부',
  name: '그랜저',
  compony: '현대자동차'
}
```

고차 함수를 이용 해서 변수 저장 하는 방법 :

```
var setAddress = setCarAddress01(address);
console.log(setAddress(car01));
console.log(setAddress(car02));
```

3-5. Currying

01. Currying

- 함수 하나가 n개의 파라미터 인자를 받는 것을 n개의 함수로 받는 방법
- 매개변수 일부를 적용하여 새로운 함수를 동적으로 생성하면 이 동적 생성된 함수는 반복적으로 사용되는 매개변수를 내부적으로 저장하여, 매번 인자를 전달하지 않아도 원본함수가 기대하는 기능을 채워 놓는다

```
var fprint = function ( a, b ) {
  return `${a} ${b}`;
};

console.log(fprint('이건', '뭐지 ....')); // 이건 뭐지 ....
```

```
var fCurryPrint = function(a) {
  return function(b) {
    return `${a} ${b}`;
  };
};

console.log(fCurryPrint('이건')('뭐지 ....')); // 이건 뭐지 ....

// 화살표 함수로 표현
var fCurryPrint = a => b => `${a} ${b}`;
```

Lexical Environment에 a를 저장하고 function(b)를 반환

```
var fAdd = function ( a, b ) {
  return a+b;
};

var fAddCurry = function(f) {
  return function(a) {
    return function(b) {
      return f(a, b);
    };
  };
};

var sum = fAddCurry(fAdd);
console.log(sum(1)(2)); // 3

// 동일한 기능을 함
fAddCurry(fAdd)(1)(2); // 3
```

4-1. 구조분해할당

01. 구조분해할당

- 배열이나 객체의 속성을 해체 하여 그 값을 변수에 담는 것

```
const arr = [1,2,3,4];

// 구조분해를 하여 각 변수에 담는다.
const [a,b,c,f] = arr; // a = 1, b = 2, c = 3, d=4
```

변수 선언 후 할당

- 구조 분해 할당 시 기본값을 지정 할 수 있음

```
var a, b;
[a, b] = [1]; // a=1, b= undefined
[a=5, b=7] = [1]; // a=1, b=7
[a, b] = [b,a] // a=7, b=1
```

함수 결과 할당

- 구조 분해 할당 시 기본값을 지정 할 수 있음

```
function funA(val) {
  return val;
}
[a, b, c ] = funA([1,2,3]); // a=1, b=2, c=3
[d, , e ] = funA([5,6,7]); // d=5, e=7, 6은 버림
[f, ...g] = funA([5,6,7]); // f=5, g=[6,7]
```

```
var o = {p: 42, q: true};
var {p, q} = o; // p=42, q=true
```

```
var {a=5, b=6} = {a:1}; // a=1, b=6
```

```
var {c: aa = 10, d: bb = 5} = {c: 3}; // aa=3, bb=5
```

5-1. This

01. 함수

- JavaScript의 this는 해당 함수 호출 패턴에 따라서 의미가 다르다.
- 전역 객체를 참조 (Global Scope)
- 메소드 내부의 this는 해당 메소드를 호출한 부모 객체를 참조
- 생성자 함수 코드 내부의 this는 새로 생성된 객체를 참조.
- call()과 apply() 메소드로 함수를 호출할 때, 함수의 this는 첫 번째 인자로 넘겨받은 객체를 참조
- 프로토타입 객체 메소드 내부의 this도 해당 메소드를 호출한 부모 객체를 참조.
- JavaScript의 this 키워드는 접근제어자 public 역할

Global Scope

- JavaScript의 모든 전역 변수는 실제로 전역 객체(브라우저에서는 window객체)의 프로퍼티들입니다.
- 모든 프로그램은 global code의 실행에 의해서 시작하고 this는 정해진 실행 문맥의 안에 고정되기 때문에 JavaScript의 this는 global code의 전역 객체입니다.

부모 객체 참조

- Counter 객체가 increment 메소드 호출 (this 의 범위는 자신을 호출한 객체 참조)

```
const counter = {
  val: 0,
  increment: function() {
    this.val += 1;
  }
};

counter.increment();
console.log(counter.val); // 1
counter['increment']();
console.log(counter.val); // 2
```

```
let varValue = 100;
const counter = {
  varValue: 0,
  increment: function() {
    console.log(this);
    this.varValue += 1;
  }
};

counter.increment();
console.log(counter.varValue); // 1

// 전역화
const inc = counter.increment;
inc();
console.log(varValue); // 101
```

```
{ varValue: 0, increment: [Function: increment] }
1
<ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  }
}
100
```

5-1. This

```
val = 100;
var counter = {
  val: 1,
  func1: function() {
    console.log("func1() :: ", this);
    this.val += 1;
    console.log('func1() this.val :: ', this.val); // func1() this.val: 2
    func2 = function() {
      console.log("func2() :: ", this);
      this.val += 1;
      console.log('func2() this.val: ' + this.val); // func2() this.val: 101
      func3 = function() {
        console.log("func3() :: ", this);
        this.val += 1;
        console.log('func3() this.val: ' + this.val);
        // func3() this.val: 102
      }
      func3();
    }
    func2();
  }
};

counter.func1();
```

JavaScript에서는 내부 함수 호출 패턴을 정의해 놓지 않기 때문
-> 내부 함수도 결국 함수이므로 이를 호출할 때는 함수 호출로 취급 됨
-> 함수 호출 패턴 규칙에 따라 내부 함수의 this는 전역 객체를 참조
-> func1(메소드)에서는 메소드 내부의 val 값이 출력,
func2(내부 함수), func3(내부 함수)에서는 전역 변수의 val 값이 출력

```
func1() :: { val: 1, func1: [Function: func1] }
func1() this.val :: 2
func2() :: <ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  val: 100,
  func2: [Function: func2]
}
func2() this.val: 101
func3() :: <ref *1> Object [global] {
  global: [Circular *1],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(nodejs.util.promisify.custom)]: [Getter]
  },
  val: 101,
  func2: [Function: func2],
  func3: [Function: func3]
}
func3() this.val: 102
```

5-2. This – 생성자 함수 호출

생성자 함수 호출

- 생성자 함수를 호출할 때, 생성자 함수 코드 내부의 this는 새로 생성된 객체를 참조

```
function F(v) {
  console.log(this); // F {}
  this.val = v;
}
var f = new F("constructor function");
console.log(f);      // F { val: 'constructor function' }
console.log(f.val);  // constructor function
console.log(val);    // ReferenceError: val is not defined
```

call과 apply Method 함수 호출

- JavaScript에는 이러한 내부적인 this 바인딩 이외에도 this를 특정 객체에 명시적으로 바인딩 시키는 apply()와 call() 메소드 제공.
- apply()와 call() 메소드로 함수를 호출할 때, 함수는 첫 번째 인자로 넘겨받은 객체를 this로 바인딩하고, 두 번째 인자로 넘긴 값들은 함수의 인자로 전달
- apply() 메소드는 배열 형태로 인자를 넘기고, call() 메소드는 배열 형태로 넘긴 것을 각각 하나의 인자로 넘기는 것입니다.

```
var add = function(x, y) {
  console.log("x ::" , x);
  console.log("y ::" , y);
  console.log("this.val :: ", this.val);
  // 첫번째 apply에서 호출 되는 경우 0으로 초기화 함 :
  // obj에 val이 정의 되어 있으므로 ) obj가 this

  this.val = x + y;
  this.val += this.hval;
};

var obj = {
  val: 0,
  hval : 10
};

add.apply(obj, [2, 8]);
console.log(obj.val); // 20
console.log("=====");
add.call(obj, 2, 8);
console.log(obj.val); // 20
```

x :: 2
y :: 8
this.val :: 0
20
=====

x :: 2
y :: 8
this.val :: 20
20

5-3. This - 프로토타입

프로토타입 객체 메소드의 this로 해당 메서드 호출

- 프로토타입 객체는 메소드를 가질 수 있다.
- 이 프로토타입 메소드 내부에서의 this도 똑같이 메소드 내부의 this는 해당 메소드를 호출한 부모 객체를 참조.

```
function Person(name) {
  console.log(" Person  This :: " , this);
  this.name = name;
}
```

```
Person.prototype.getName = function() {
  console.log("This :: " , this);
  return this.name;
}
```

```
var fPerson = new Person("홍길동"); // this :: Person {}
console.log(fPerson.getName());      // this :: Person { name: '홍길동' } -> 체이닝 발생 :: 홍길동
```

```
Person.prototype.name = "홍당무";
console.log(fPerson.getName());      // this :: Person { name: '홍길동' } -> 체이닝 발생 :: 홍길동
console.log(Person.prototype.getName()); // this :: { getName: [Function (anonymous)], name: '홍당무' } :: 홍당무
```

- getName() 메소드는 foo 객체에서 찾을 수 없으므로 프로토타입 체이닝이 발생
- fPerson 객체의 프로토타입 객체인 person.prototype에서 getName() 메소드가 있으므로, 이 메소드가 호출
- getName() 메소드를 호출한 객체는 fPerson 이므로, this는 foo 객체에 바인딩
- fPerson.getName()의 결과로 홍길동 가 출력

Public 역할

- public, private 키워드 자체를 지원하지 않지만 public 역할을 하는 this 키워드와 private 역할을 하는 var 키워드가 있습니다. JavaScript에서는 이 두 키워드를 사용하여 캡슐화

```
function Person(name, age) {
  let Name = name;
  this.Age = age;
  this.getName = function() {
    return Name;
  }
}
```

```
var fPerson = new Person('홍당무', 26);
console.log(fPerson.Age);      // 26
console.log(fPerson.Name);     // undefined 내부 변수로 접근 불가
console.log(fPerson.getName()); // 홍당무
```


5-4. This 함정

5. this

This 함정

fTest 에서 this가 Func를 가리킬 것으로 생각되지만. 실제로는 그렇지 않다. -----> fTest에서 Func에 접근하려면 method에 Local 변수를 하나 만들고 Func를 가리키게 하여야 한다

```
var Func = {
  name : "홍당무"
};
Func.method = function() {
  function fTest() {
    console.log("this= ", this); // Object [global] {.....
    console.log("this.name = ", this.name); // undefined
  };

  fTest();
}

Func.method();
```

```
var Func = {
  name : "홍당무"
};
Func.method = function() {
  let self = this; // Local 변수를 통한 접근
  function fTest() {
    console.log("self = ", self); // { method: [Function], name: '홍길동' }
    console.log("self.name = ", self.name); // 홍길동
  };
  fTest();
}

Func.method();
```

콜백 함수에 this에 다른 값을 전달 하는 방법

```
var ObjA = {
  values: [1,2,3],
  fPrintValues: function(value, index) {
    var self = this;
    console.log(self);
    // { values: [ 1, 2, 3 ], fPrintValues: [Function: fPrintValues] }
    return function(value, index) {
      console.log(this); // global
      return console.log(self.values, value, index);
    }
  }
}

var fCb = ObjA.fPrintValues();
console.log(`${fCb}`);
// function(value, index) {
//   console.log(this); // global
//   return console.log(self.values, value, index);
// }
```

call : fCb(10,10); // [1, 2, 3] 10 10
-> global 에서 실행이 되었으므로 this는 global 이다.

Call : [10,20,30].forEach(fCb);
// [1, 2, 3] 10 0
// [1, 2, 3] 20 1
// [1, 2, 3] 30 2

5-4. This 함정

This 함정

ECMAScript 5부터는 익명 함수와 결합된 bind 메서드를 사용하여 같은 결과를 얻을 수 있다.

```
var Func = {
  name : "홍당무"
};

Func.method = function() {
  var ftest = function () {
    console.log("this = ", this);
    console.log("this.name = ", this.name);
  }.bind(this);

  ftest();
}

Func.method();
```

```
var objA = {
  name: 'OBJ A',
  func: function() {
    console.log(this.name);
  }
}

objA.func(); // OBJ A
setTimeout(objA.func, 100); // undefined
setTimeout(objA.func.bind(objA), 100); // OBJ A
var objB = { name: 'OBJ B' };
setTimeout(objA.func.bind(objB), 100); // OBJ B
```

프로토 타입 상속 (prototypal inheritance)

```
function Func01() {};
Func01.prototype.name = "홀길동";

Func01.prototype.method = function() {
  console.log("this ::", this);
  console.log("this.name ::", this.name);
};
new Func01().method(); // this :: Func01 {}, this.name :: 홀길동

function Func02() {}
Func02.prototype = Func01.prototype; // 프로토 타입 상속

var fFunc02= new Func02();
fFunc02.method(); // this :: Func01 {}, this.name :: 홀길동
```

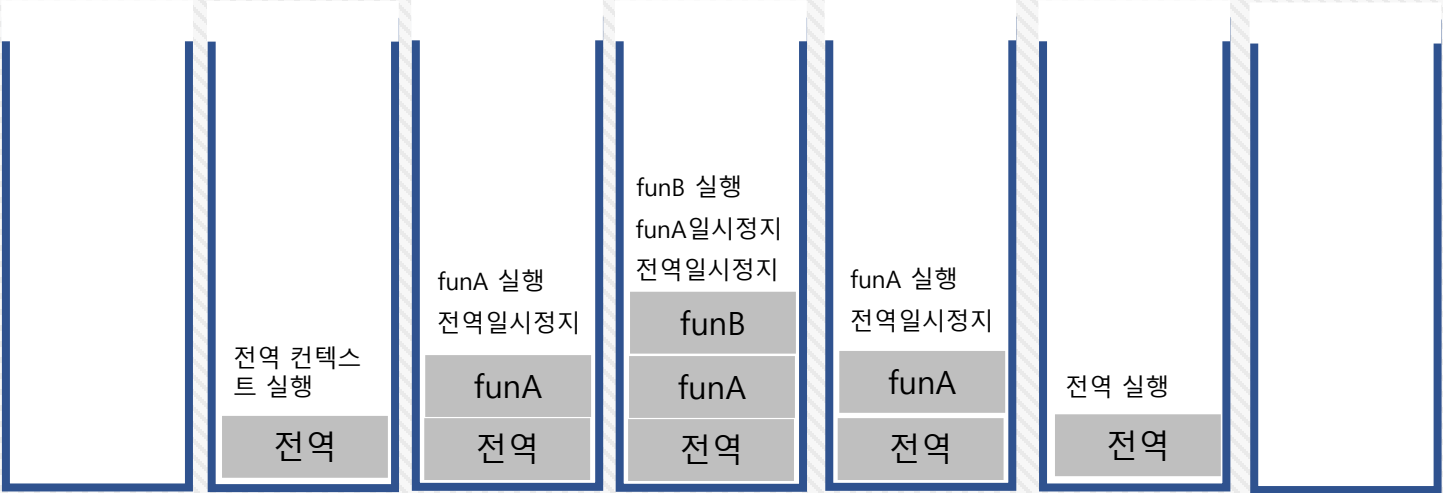
01. 실행 컨텍스트

- 실행할 코드에 제공할 환경 정보를 모아놓은 객체
- 함수의 환경 정보를 수집 해서 실행 순서 call stack을 통해서 구성 한다.

```
var a = 1;

function funA() {
  function funB() {
    console.log("Run FunB a :: ", a);
    var a = 3;
  }
  funB();
  console.log("Run funA a :: ", a);
}

funA();
console.log("Run funA After a :: ", a);
```



funB	VariableEnvironment : 현재 컨텍스트 내의 식별자 정보 (environmentRecord (snapshot) + 외부환경정보 (outerEnviromentReference (snapshot)) => 변경 사항은 변경 되지 않음
	LexicalEnvironment : 현재 컨텍스트 내의 식별자 정보 (environmentRecord)+ 외부환경정보 (outerEnviromentReference) => 변경 사항은 변경 됨
	ThisBinding : this 식별자가 바라봐야 하는 대상 객체
funA	
전역	

6-2. Scope

01. Scope

- 변수와 매개변수의 접근성과 생존 시간을 의미 함
- 전역범위(Global Scope)와 지역 유효 범위(Local Scope)로 나뉜다.
- 전역범위는 전체를 의미 하며 지역 범위는 정의 된 함수 내에서만 참조 되는 것을 의미 함 (Function-Level Scope)
- 다른 언어는 Block-level Scope 를 사용 하는데 ES6에 등장한 let, const를 사용 하면 block-level scope를 사용 할 수 있다.

변수 범위

```
var gVal = "global"; // 전역 Scope

function fFuncLevel01() {
  var fLevel01Val = "1 Level";
  console.log("gVal ::", gVal); // gVal :: global
  fFuncLevel02();
  function fFuncLevel02() {
    console.log("fLevel01Val ::", fLevel01Val); // 1 Level
    var fLevel02Val = "2 Level";
    console.log("fLevel02Val ::", fLevel02Val); // 2 Level
  }
  //console.log("fLevel02Val ::", fLevel02Val);
  // => ReferenceError: fLevel02Val is not defined
}

fFuncLevel01();
// console.log("fLevel01Val ::", fLevel01Val);
// => ReferenceError: fLevel02Val is not defined
```

- gVal : 전역 범위
- fLevel01Val : 함수 fFuncLevel01 내부
- fLevel02Val : 함수 fFuncLevel02 내부

함수 단위의 유효 범위 (비 블록 레벨 스코프(Non block-level scope))

```
function fFunc() {
  var a = 0;
  if (true) {
    var b = 0;    // 비 블록 레벨 스코프(Non block-level scope)
    for ( var c = 0; c < 5; c++) {
      // 작업
    }
    console.log("c=",c); // c= 5
  }

  console.log("b=",b); // b= 0
}

fFunc();
```

변수명 중복

```
var gVal = "global";
function fFunc() {
  var gVal = "local";
  console.log("gVal =", gVal); // gVal = local
}

fFunc();
console.log("gVal =", gVal); // gVal = global
```

6-2. Scope

var 키워드 생략 (암묵적 전역)

```
function fFunc() {  
    gVal = "local";  
}  
  
function fFunc01() {  
    console.log("gVal =", gVal); // local  
}  
fFunc();  
fFunc01();
```

렉시컬 특성

- 렉시컬 스코프는 함수를 어디서 호출하는지가 아니라 어디에 선언하였는지에 따라 결정된다

```
function f1(){  
    var a = 10; f2();  
}  
function f2(){  
    console.log("호출 ");  
}  
f1(); // 결과 10  
function f1(){  
    a = 10; console.log(f2());  
}  
function f2(){  
    return a;  
}  
f1(); // 결과 10
```

```
var x = 1;  
function foo() { // 전역에 선언  
    var x = 10;  
    bar();  
}  
function bar() { // 전역에 선언  
    console.log(x);  
}  
foo(); // 결과 : 1  
bar(); // 결과 : 1
```

최소한의 전역변수 사용

- 애플리케이션에서 전역변수 사용을 위해 다음과 같이 전역변수 객체 하나를 만들어 사용하는 것이다. (더글라스 크락포드의 제안)

```
var MyApp = {};  
MyApp.Name = {  
    firstName: "홍",  
    secondName: "당무"  
}  
  
MyApp.Print = function() {  
    console.log("MyApp :: \n", MyApp) ;  
    // MyApp ::  
    // { Name: { firstName: '홍', secondName: '당무' },  
    //   Print: [Function] }  
    console.log("Name :: ", MyApp.Name.firstName + MyApp.Name.secondName) ;  
    // Name :: 홍당무  
}  
  
MyApp.Print();
```

즉시실행함수를 이용한 전역변수 사용 억제

- 즉시 실행 함수는 즉시 실행되고 그 후 전역에서 바로 사라진다

```
(function () {  
    var MYAPP = {};  
    MYAPP.student = {  
        name: '홍당무',  
        gender: '남자'  
    };  
    console.log(MYAPP.student.name);  
})();  
  
console.log(MYAPP.student.name); // ReferenceError: MYAPP is not defined
```

6-2. Scope Chain

01. Scope Chain

- 함수 외부에 선언한 변수는 함수 내부에서도 사용 가능 하지만 함수 내부에 선언한 변수는 함수 내부에서만 가능 하다.
- Scope Chain은 식별자의 유효 범위를 함수 내부에서 외부로 차례로 검색 하는 것이다.

```
var a = 1;
var funA = function() {
  var funB = function() {
    console.log(a); // undefined
    var a = 2; // 변수 은닉화
  };
  funB();
  console.log(a); // 1
}

funA();
console.log(a); // 1
```

전역 컨텍스트 실행

전역 : L.E
- E.R : { a, funcA }
- This : global

funA : L.E
- E.R : { funB }
- O.E.R : [global, { a, funcA }]
- This : global

전역 : L.E
- E.R : { a, funcA }
- This : global

funA 실행 이후 a는 외부에 있는 변수 a를 찾아 출력 함 // 1

funB : L.E
- E.R : { a }
- O.E.R : [funcA, { funB }]
- This : global

funA : L.E
- E.R : { funB }
- O.E.R : [global, { a, funcA }]
- This : global

전역 : L.E
- E.R : { a, funcA }
- This : global

funB 실행 시 a는 hosting되어 선언만 된 상태 이므로 // undefined

6-3. Hosting

02. Hosting

- 모든 변수는 함수 본문 어느 부분에서 선언 되더라도 내부적으로 함수의 맨 윗부분으로 끌어올려진다.
- 함수도 변수에 할당 되는 객체이기 때문에 동일하게 적용이 된다.

변수 Hosting

```
function funA(x) {
  console.log(x)    // 1
  var x;
  console.log(x)    // 1
  var x = 10;
  console.log(x)    // 10
}

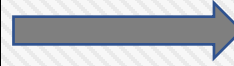
funA(1);
```



파라미터는 변수 선언된 것으로 인식

```
function funA(x) {
  var x = 1;
  console.log(x)    // 1
  var x;
  console.log(x)    // 1
  var x = 10;
  console.log(x)    // 10
}

funA(1);
```



Hosting

```
function funA(x) {
  var x = 1;
  var x;
  var x;
  x = 1;
  console.log(x)    // 1
  console.log(x)    // 1
  x = 10;
  console.log(x)    // 10
}

funA(1);
```

```
console.log("변수 :: a = ", a); // 변수 :: a = undefined
var a = 1; ← Hosting
let b = 1;
const c = 1;
console.log("a :: "+ a , " b :: "+ a, ", c :: "+ c);
// a :: 1 b :: 1 , c :: 1
```

6-3. Hosting

02. Hosting

- 함수는 전체가 Hoisting 된다
- 함수 표현식은 식별자만 hosting 된다.

함수 Hoisting

```
function funcA() {  
  console.log(a); // [Function: a]  
  var a = 1;  
  console.log(a); // 1  
  function a() {}  
}
```

funcA();



Hoisting

```
function funcA() {  
  var a;  
  function a() {};  
  
  console.log(a); // [Function: a]  
  a = 1;  
  console.log(a); // 1  
}
```

funcA();

```
function funcA() {  
  console.log(a); // undefined  
  var a = 1;  
  console.log(a); // 1  
  var a = function() {};  
}
```



Hoisting

```
function funcA() {  
  var a ;  
  var a ;  
  
  console.log(a); // undefined  
  a = 1;  
  console.log(a); // 1  
  a = function() {}  
}
```

}

funcA();

6-3. Hosting

02. Hosting

- 모든 변수는 함수 본문 어느 부분에서 선언 되더라도 내부적으로 함수의 맨 윗부분으로 끌어올려진다.
- 함수도 변수에 할당 되는 객체이기 때문에 동일하게 적용이 된다.

함수 Hosting

```
function fFunc01() {
  console.log("gloabl fFunc01() ... ");
}

function fHoisting() {
  console.log(typeof fFunc01, " :: ", fFunc01);
  // function :: function fFunc01() {
  //           console.log("gloabl fFunc01() ... ");
  //           }
  console.log(typeof fFunc02, " :: ", fFunc02); ← Hosting
  // function :: function fFunc01() ...
  fFunc01();
  fFunc02();
}

fHoisting();

function fFunc02() {
  console.log("gloabl fFunc02() ... ");
}
```

함수 Hosting -> 함수 표현식은 hoisting 되지만 정의된 함수는 hoisting 되지 않는다.

```
function fFunc01() {
  console.log("gloabl fFunc01() ... ");
}

function fHoisting() {
  console.log(typeof fFunc01); // function :: function fFunc01 ← Hosting
  console.log(typeof fFunc02); // undefined ← Hosting
  console.log(typeof fFunc03); // undefined ← Hosting
  console.log(typeof fFunc04); // function :: function fFunc04 ← Hosting
  console.log(typeof fFunc05); // function :: function fFunc05 ← Hosting
  fFunc01(); // local fFunc01() ...
  // fFunc02(); // TypeError: fFunc02 is not a function
  // fFunc03(); // TypeError: fFunc03 is not a function
  fFunc04(); // local fFunc04() ...
  fFunc05(); // gloabl fFunc05() ...
  function fFunc01() { // 재정의 -> 유효범위 체인
    console.log("local fFunc01() ... ");
  }
  var fFunc02 = function() { // 재정의 -> 유효범위 체인
    console.log("local fFunc02() ... ");
  }
  var fFunc03 = function () {
    console.log("local fFunc03() ... ");
  }
  function fFunc04() {
    console.log("local fFunc04() ... ");
  }
  console.log(typeof fFunc02, " :: ", fFunc02); // function :: function() {
  fFunc02(); // local fFunc02() ...
}

fHoisting();
function fFunc02() {
  console.log("gloabl fFunc02() ... ");
}
function fFunc05() {
  console.log("gloabl fFunc05() ... ");
}
```

6-4. Closure

03. Closure

- JavaScript의 유효범위 체인을 이용하여 이미 생명 주기가 끝난 외부 함수의 변수를 참조하는 방법
- 외부에서 내부 변수에 접근할 수 있도록 하는 함수입니다
- 클로저를 만들면 클로저 스코프 안에서 클로저를 만든 외부 스코프(Scope)에 항상 접근할 있다

```
function counter(start) {
  var count = start;
  return {
    increment: function() {
      count++;
    },
    get: function() {
      return count;
    }
  };
}
var foo = new counter(4);
foo.increment();
console.log(foo.get()); // 5

foo.hack = function() {
  count = 1337;
};
foo.hack();
console.log(foo.get()); // 5
console.log(count); // 1337
```

- counter는 increment, get 두개의 Closure를 가지고 있음.
- 두개의 함수가 count를 참조 하고 있으므로 이 함수 Scope안에서는 count 변수를 계속 접근 할 수 있음
- hack 함수는 Counter 함수에 정의 가 되어 있지 않아서 count에 설정을 하여도 count 값은 변경 되지 않는다. Hack에서 변경 한 count 는 Global 변수에 적용 됨

반복문 에서 Closure 사용 하기

- 타이머에 설정된 익명 함수는 변수 i에 대한 참조를 들고 있다가 console.log가 호출되는 시점에 i의 값을 사용한다.
- console.log가 호출되는 시점에서 for loop는 이미 끝난 상태기 때문에 i 값은 10이 된다

```
for(var i = 0; i < 10; i++) {
  setTimeout(function() {
    console.log(i); // 10 ( 10만 출력됨 )
  }, 100);
}
```

1. 익명 함수로 래핑

```
for(var i = 0; i < 10; i++) {
  ((e)=> { // function(e) {
    setTimeout(function() {
      console.log(e);
      // 0,1,2,3,4,5,6,8,9
    }, 100);
  })(i);
}
```

3. setTimeout 함수에 세번째 인자를 추가

```
for(var i = 0; i < 10; i++) {
  setTimeout(function(e) {
    console.log(e);
    // 0,1,2,3,4,5,6,8,9
  }, 100, i);
}
```

2. 래핑한 익명 함수에서 출력 함수를 반환

```
for(var i = 0; i < 10; i++) {
  setTimeout((function(e) {
    return function() {
      console.log(e);
      // 0,1,2,3,4,5,6,8,9
    }
  })(i), 100);
}
```

4. bind를 사용하는 방법

```
for(var i = 0; i < 10; i++) {
  setTimeout(
    console.log.bind(console, i),
    100);
  // 0,1,2,3,4,5,6,8,9
}
```

6-4. Closure

Closure 오남용

- 함수 내부의 클로저 구현은 함수의 객체가 생성될 때마다 클로저가 생성되는 결과를 가져옵니다. 같은 구동을 하는 클로저가 객체 마다 생성이 된다면 쓸데없이 메모리를 쓸데없이 차지하게 되는데, 이를 클로저의 오남용이라함 -> 성능 문제 (메모리)

```
function MyObject(inputname) {  
  this.name = inputname;  
  this.getName = function() { // 클로저  
    return this.name;  
  };  
  this.setName = function(rename) { // 클로저  
    this.name = rename;  
  };  
}  
  
var obj = new MyObject("서");  
console.log(obj.getName());
```

→
prototype객체를 이용한
클로저 생성

```
function MyObject(inputname) {  
  this.name = inputname;  
}  
  
MyObject.prototype.getName = function() {  
  return this.name;  
};  
MyObject.prototype.setName = function(rename) {  
  this.name = rename;  
};  
  
var obj = new MyObject("서");  
console.log(obj.getName());
```

Closure를 통해서는 외부 함수의 this와 arguments객체를 참조 할 수 없음

```
function f1(){  
  console.log(arguments[0]); // 1  
  function f2(){  
    console.log(arguments[0]); // undefined  
  }  
  return f2;  
}  
  
var exam = f1(1);  
exam();
```

→

```
function f1(){  
  console.log(arguments[0]); // 1  
  var a = arguments[0];  
  function f2(){  
    console.log(a); // 1  
  }  
  return f2;  
}  
  
var exam = f1(1);  
exam();
```

6-4. Closure

Closure을 이용한 캡슐화

- 캡슐화란 간단하게 말하면 객체의 자료화 행위를 하나로 묶고, 실제 구현 내용을 외부에 감추는 겁니다. 즉, 외부에서 볼 때는 실제 하는 것이 아닌 추상화 되어 보이게 하는 것으로 정보은닉

```
function Gugudan(dan){
  this.maxDan = 3;
  this.calculate = function(){
    for(var i =1; i<=this.maxDan; i++){
      console.log(dan+"*"+i+"="+dan*i);
    }
  }
  this.setMaxDan = function(reset){
    this.maxDan = reset;
  }
}

var dan5 = new Gugudan(5);
dan5.calculate();
dan5.maxDan=2;
dan5.calculate();
dan5.setMaxDan(4);
dan5.calculate();
```

5*1=5
5*2=10
5*3=15

5*1=5
5*2=10

5*1=5
5*2=10
5*3=15
5*4=20

7-1. argement 객체

01. argement 객체

- JavaScript의 모든 함수 스코프에는 arguments라는 특별한 변수가 있다. 이 변수는 함수에 넘겨진 모든 인자에 대한 정보가 담겨 있다
- arguments 객체는 Array가 아니다. 물론 length 프로퍼티도 있고 여러모로 Array와 비슷하게 생겼지만 Array.prototype을 상속받지는 않았다
- arguments에는 push, pop, slice 같은 표준 메소드가 없다.
- 일반 for문을 이용해 순회는 할 수 있지만, Array의 메소드를 이용하려면 arguments를 Array로 변환해야 한다.
- arguments 변수는 Function 안에서 다시 정의할 수 없다.
- var 구문이나 파라미터에 arguments라는 이름으로 변수를 정의해도 변수가 재정의되지 않는다

```
function Person(first, last) {
  this.first = first;
  this.last = last;
}

Person.prototype.fullname = function(joiner, options) {
  options = options || { order: "western" };
  var first = options.order === "western" ? this.first : this.last;
  var last = options.order === "western" ? this.last : this.first;
  return first + (joiner || " ") + last;
};

Person.fullname = function() {
  // 결과: Person.prototype.fullname.call(this, joiner, ..., argN);
  return Function.call.apply(Person.prototype.fullname, arguments);
};

var grace = new Person("Grace", "Hopper");
console.log(grace.fullname()); // 'Grace Hopper'
console.log(Person.fullname({ first: "Alan", last: "Turing" }, ", ", { order: "eestern" }));
console.log(Person.fullname({ first: "Alan", last: "Turing" }, ", "));
```

"fullname" 메서드의 비결합(unbound) 버전을 생성한다. 첫번째 인자로 'first'와 'last' 속성을 가지고 있는 어떤 객체도 사용 가능하다.
"fullname"의 인자 개수나 순서가 변경되더라도 이 래퍼를 변경할 필요는 없을 것이다.

8-1. JavaScript Object prototype

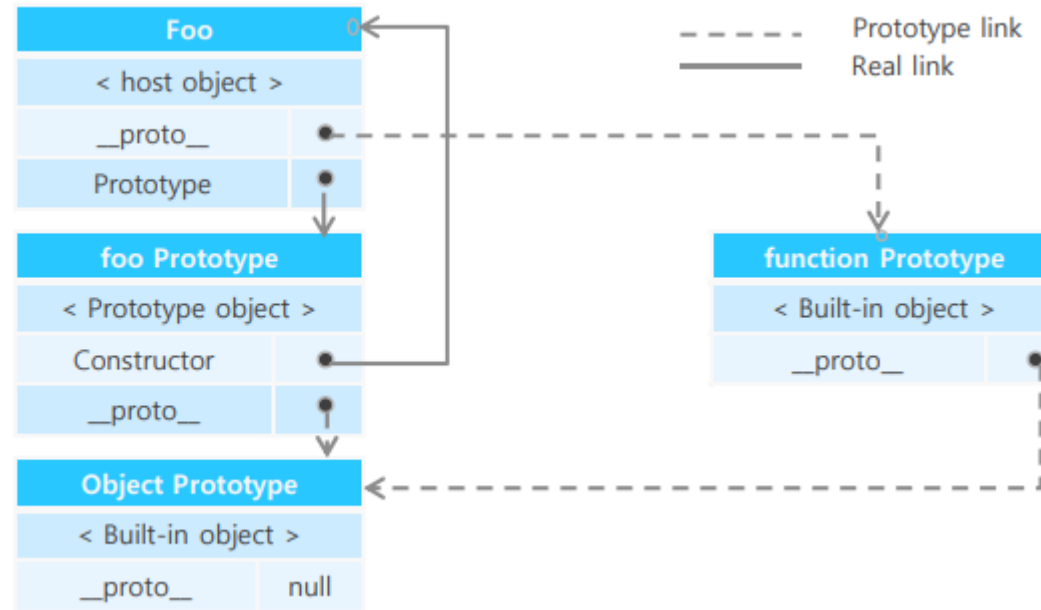
01. JavaScript Object prototype 란

- 객체의 원형인 프로토타입을 이용하여 새로운 객체를 만드는 프로그래밍 기법
- JavaScript의 모든 객체는 자신을 생성한 객체 원형에 대한 숨겨진 연결을 갖는다. 이때 자기 자신을 생성 하기 위해 사용된 객체원형을 프로토타입이라 함
- JavaScript의 모든 객체는 Object객체의 프로토타입을 기반으로 확장 되었기 때문에 이 연결의 끝은 Object객체의 프로토타입 Object임

Prototype property 란 ?

- 모든 함수 객체의 Constructor는 prototype이란 프로퍼티를 가지고 있다.
- 이 prototype property는 Object가 생성될 당시 만들어지는 객체 자신의 원형이 될 prototype 객체를 가리킨다.

```
function foo() {}
var foo = new foo();
```



8-1. JavaScript Object prototype

예제 1

```
function foo(x) {  
    this.x = x;  
};  
var A = new foo('hello');  
console.log(A.x); // hello  
console.log(foo.prototype.x); // undefined  
console.log(A.prototype.x) // Cannot read property 'x' of undefined
```

- A 는 함수 객체가 아닌 foo라는 원형 함수 객체를 통해 만들어진 Object 객체에 확장된 단일 객체
- A 객체가 생성 시점에 foo의 prototype.x가 없음
-> A는 prototype property를 소유하고 있지 않음

예제 2

```
var A = function () {  
    this.x = function () {  
        console.log('hello');  
    };  
};  
A.X = function() {  
    console.log('world');  
};  
  
var B = new A();  
var C = new A();  
B.x(); // hello  
C.x(); // hello  
A.x(); // world
```

- B,C 는 객체 생성시 A의 prototype Object를 사용함
- prototype Object는 A가 생성될 시점 정보만 가지고 있음
- A의 prototype Object가 알고 있는 x는 function () { console.log('hello');};임
- A.x를 아무리 수정을 하여도 A의 prototype Object는 변경되지 않음

8-1. JavaScript Object prototype

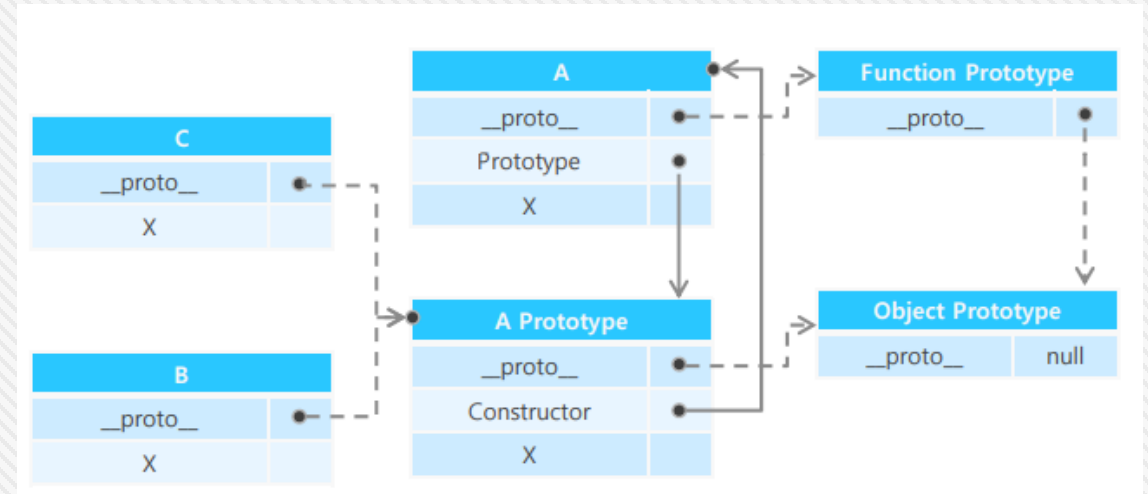
8. prototype

예제 3

```
var A = function () { };
A.x = function() {
  console.log('hello');
};
A.prototype.x = function () {
  console.log('world');
};
var B = new A();
var C = new A();
B.x(); // world
C.x(); // world
A.x(); // hello

A.prototype.x = function () {
  console.log('world2');
};
B.x(); // world2
C.x(); // world2
```

- B,C 는 객체 생성시 A의 prototype Object를 사용함
- prototype Object는 A가 생성될 시점 정보만 가지고 있음
- A의 prototype.x를 world로 선언 하였으므로 B,C는 A의 prototype을 따라감



객체의 생성 과정에서 모태가 되는 프로토타입과의 연결고리가 이어져 상속관계를 통하여 상위 프로토타입으로 연속해서 이어지는 관계를 프로토타입 체인이라고 한다. 이 연결은 `__proto__` 를 따라 올라가게 된다

8-2. 공유와 상속

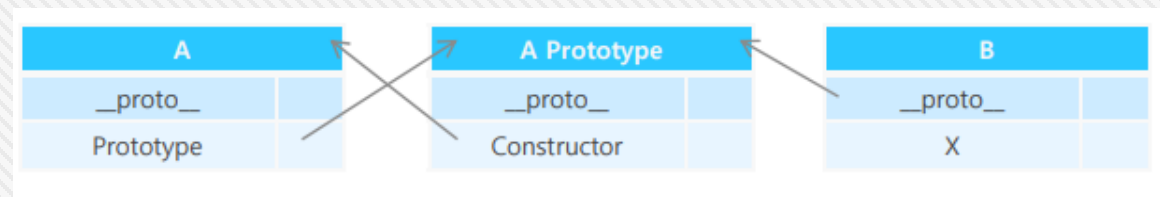
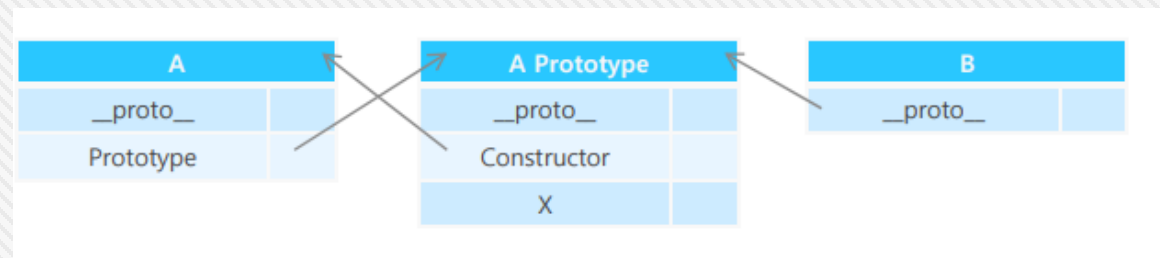
8. prototype

공유와 상속

```
// B는 x를 상속
var A = function() {};
var B = new A();
A.prototype.x = 'hello';
console.log(A)           // [Function: A]
console.log(B)           // A {}
console.log(A.prototype.x); // hello
console.log(B.x);        // hello

B.x = 'world';
console.log(A) ;         // [Function: A]
console.log(B) ;         // A { x: 'world' }
console.log(A.prototype.x); // hello
console.log(B.x); // world

// B는 x를 공유
var C = function() {
  this.x = 'hello';
};
var D = new C();
console.log(D);           // { x: 'hello' }
console.log(C);           // [Function: C]
console.log(D.x);         // hello
console.log(C.x);         // undefined
```



8-3. 객체와 Prototype 차이점

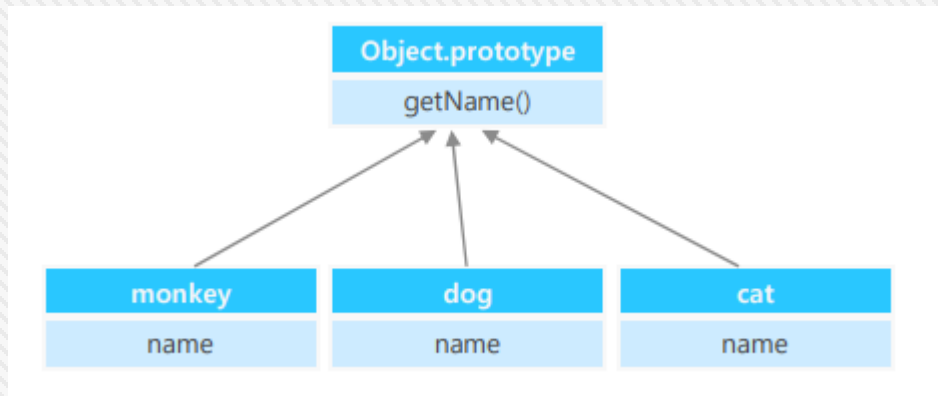
8. prototype

객체와 Prototype이 차이점

```
function Animal(name) {  
  this.name = name;  
  this.getName = function () {  
    console.log(this.name);  
  }  
}  
var monkey = new Animal("Monkey");  
var dog = new Animal("dog");  
var cat = new Animal("cat");  
monkey.getName();           // Monkey  
dog.getName();              // dog  
cat.getName();              // cat
```

```
Object.prototype.getName = function () {  
  console.log(this.name);  
}  
function Animal(name) {  
  this.name = name;  
}  
var monkey = new Animal("Monkey");  
var dog = new Animal("dog");  
var cat = new Animal("cat");  
monkey.getName();           // Monkey  
dog.getName();              // dog  
cat.getName();              // cat
```

monkey	dog	cat
name	name	name
getName()	getName()	getName()



01. Class

- class 키워드 선언 하고 new 키워드로 사용함.
- constructor 함수를 사용하여 멤버 속성을 정의함 (this 사용)
- 모든 속성은 공개 속성임
- 메소드는 function을 사용 하지 않는다
- 자바 스크립트의 Class도 Prototype 이다
- Class를 사용 하면 간단히 사용 할 수 있다

```
// ES 2015 이전
function Car (name, compony) {
  this.name = name;
  this.compony = compony;
};
Car.prototype.getName = function(){
  return `${this.name}`;
};
Car.prototype.setName= function(name) {
  this.name = name;
};
Car.prototype.getCar= function(name) {
  return this;
};

const car = new Car('소나타', '현대');

console.log(car.getCar());
console.log(car);
console.log(car.getName());
console.log(car.compony);
```

```
class Car {
  // 생성자
  constructor(name, compony) {
    // this 를 사용 해서 속성 설정
    this.name = name;
    this.compony = compony;
  }

  getName() {
    return `${this.name}`;
  }

  setName(name) {
    this.name = name;
  }

  getCar() {
    return this ;
  }
}
```

```
class CarAtt extends Car {
  constructor(name, compony, color) {
    super(name, compony);
    this.color = color;
  }

  getCar() {
    return { ...super.getCar(),
      color: this.color };
  }

  getCarThis() {
    return this ;
  }
}
```

```
const carAtt = new CarAtt('소나타', '현대', '검정');

console.log(carAtt.getCar());
// { name: '소나타', compony: '현대', color: '검정' }
console.log(carAtt.getCarThis());
// CarAtt { name: '소나타', compony: '현대', color: '검정' }
console.log(carAtt);
// CarAtt { name: '소나타', compony: '현대', color: '검정' }
console.log(carAtt.color);
// 검정
```

9-2. Class – set, get

01. set, get

- set 으로 선언 시 파라미터는 하나 임

```
class Car {
  // 생성자
  constructor(name, compony) {
    // this 를 사용 해서 속성 설정
    this.name = name;
    this.compony = compony;
  }

  getName() {
    return `${this.name}`;
  }

  setName(name) {
    this.name = name;
  }

  getCar() {
    return this ;
  }
}
```

```
class Car {
  // 생성자
  constructor(name, compony) {
    // this 를 사용 해서 속성 설정
    this.name = name;
    this.compony = compony;
  }

  get fname() {
    return `${this.name}`;
  }

  set fname(name) {
    this.name = name;
  }

  get car() {
    return this ;
  }

  setCar(name, compony) {
    this.name = name;
    this.compony = compony;
  }
}
```

```
const car = new Car('소나타', '현대');

console.log(car);
// Car { name: '소나타', compony: '현대' }

console.log(car.name);
// 소나타

car.name = '그랜저';

console.log(car.name);
// 그랜저

car.setCar('SM', '삼성');

console.log(car);
// Car { name: 'SM', compony: '삼성' }
```

9-2. Class – set, get

01. set, get

- set 으로 선언 시 파라미터는 하나 임

```
class Car {
  // 생성자
  constructor(name, compony) {
    // this 를 사용 해서 속성 설정
    this.name = name;
    this.compony = compony;
  }

  get name() {
    return this.name;
  }

  set name(name) {
    this.name = name;
  }

  get car() {
    return this ;
  }

  carChange(name, compony) {
    this.name = name;
    this.compony = compony;
  }
}

const car = new Car('그랜저', '현대');
console.log(car.name);
```

✖ ▶ Uncaught RangeError: Maximum call variable.js:14
stack size exceeded

```
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
at Car.set name [as name] (variable.js:14)
```

```
class Car {
  // 생성자
  constructor(name, compony) {
    // this 를 사용 해서 속성 설정
    this.name = name;
    this.compony = compony;
  }

  get name() {
    return this._name;
  }

  set name(name) {
    this._name = name;
  }

  get car() {
    return this ;
  }

  carChange(name, compony) {
    this.name = name;
    this.compony = compony;
  }
}

const car = new Car('그랜저', '현대');
console.log(car.name); // 그랜저
```

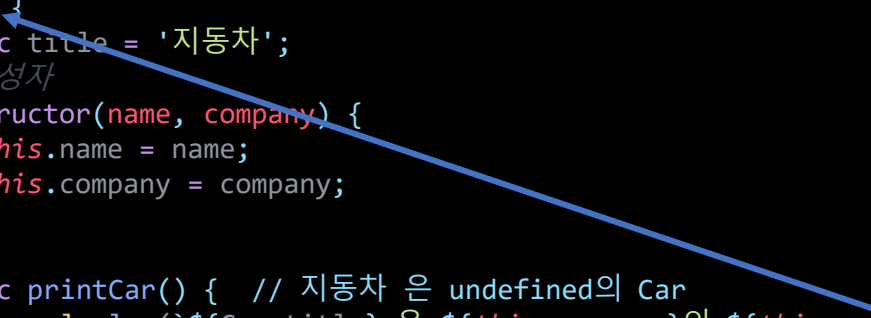
9-2. Class – public, private

01. public, private

```
class Car {  
  name = '소나타'; // public  
  #company = '현대'; // private  
}  
  
const car = new Car();  
console.log(car.name); // 소나타  
console.log(car.company); // undefined
```

02. static

```
class Car {  
  static title = '지동차';  
  // 생성자  
  constructor(name, company) {  
    this.name = name;  
    this.company = company;  
  }  
  
  static printCar() { // 지동차 은 undefined의 Car  
    console.log(`${Car.title} 은 ${this.company}의 ${this.name}`);  
  }  
  
  static sPrintCar() { // 지동차  
    console.log(`${Car.title}`);  
  }  
  
  printCar() { // 지동차 은 현대의 그랜저  
    console.log(`${Car.title} 은 ${this.company}의 ${this.name}`);  
  }  
}  
  
const hCar = new Car('그랜저', '현대');  
Car.printCar();  
Car.sPrintCar();  
hCar.printCar();  
console.log(Car.title); // 지동차
```



10-1. 콜백

콜백은 함수이며 호출 되는 시점에 제어권이 넘어 간다.

콜백 지옥은 콜백 함수를 익명 함수로 전달 하는 과정에서 반복되어 코드가 복잡 해 지는 것을 의미 한다.

콜백은 함수

```
var cnt = 0;
var fTimerCB = function() {
  if (cnt == 0 ) console.log(timer._onTimeout);
  console.log(cnt);
  if (++cnt > 4) {
    clearInterval(timer);
    console.log(timer._onTimeout);
  }
};

var timer = setInterval(fTimerCB, 300);
console.log("start");
```

```
var ObjA = {
  values: [1,2,3],
  fPrintValues: function(value, index) {
    console.log(this, value, index);
  }
}

ObjA.fPrintValues(10,10);
// { values:[ 1, 2, 3 ],fPrintValues:[Function: fPrintValues] } 10 10
[10,20,30].forEach(ObjA.fPrintValues);
// global ,10 ,0 // global,20,1 // global,30, 2
```

콜백 지옥

```
var car = '';
setTimeout(function (name) {
  car += name + ' ';
  console.log(name);
  setTimeout(function (name) {
    car += name + ' ';
    console.log(name);
    setTimeout(function (name) {
      car += name + ' ';
      console.log(name);
      console.log('Last : ' + car);
    },100, '그랜저');
  },100, 'SM5');
},100, '소나타');
```

함수로 분리

```
var car = '';
var car1 = function (name) {
  car += name + ' ';
  console.log(name);
  setTimeout(car2, 100, 'SM5');
};
var car2 = function (name) {
  car += name + ' ';
  console.log(name);
  setTimeout(car3, 100, '그랜저');
};
var car3 = function (name) {
  car += name + ' ';
  console.log(name);
  console.log('Last : ' + car);
};
setTimeout(car1, 100, '소나타');
```

함수로 분리 – promise

```
var car = '';
var carPrint = function(name) {
    return new Promise(function(resolve) {
        setTimeout(function(){
            car += name + ' ';
            resolve(name);
        },100);
    }).then( function(name) {
        console.log( name);
        return new Promise(function(resolve) {
            name = 'SM5';
            setTimeout(function(){
                car += name + ' ';
                resolve(name);
            },100);
        });
    }).then( function(name) {
        console.log( name);
        return new Promise(function(resolve) {
            name = '그랜저';
            setTimeout(function(){
                car += name + ' ';
                resolve(name);
            },100);
        });
    }).then(function(name) {
        console.log( name);
        console.log('Last : ' + car);
    });
}

carPrint('소나타');
console.log('start');
```

```
var car = '';
var car1 = function (name) {
    return new Promise(function(resolve) {
        setTimeout(function(){
            car += name + ' ';
            console.log(name);
            resolve();
        },100);
    });
};
var car2 = function (name) {
    return new Promise(function(resolve) {
        setTimeout(function(){
            car += name + ' ';
            console.log(name);
            resolve();
        },100);
    });
};
var car3 = function (name) {
    return new Promise(function(resolve) {
        setTimeout(function(){
            car += name + ' ';
            console.log(name);
            resolve();
        },100);
    });
};

car1('소나타')
    .then(car2('SM5'))
    .then(car3('그랜저'))
    .then(function() {
        console.log('Last : ' + car);
    });
console.log('start');
```

```
var car = '';
var fWait = function (name) {
    return new Promise(function(resolve) {
        setTimeout(function(){
            resolve(name);
        },100);
    });
};

var fprint = async function(name) {
    let nn = await fWait(name);
    car += name + ' ';
    return nn;
}

var carPrint = async function() {
    let nn = await fprint('소나타');
    console.log(nn);
    nn = await fprint('SM5');
    console.log(nn);
    nn = await fprint('그랜저');
    console.log(nn);
    console.log('Last : ' + car);
}

carPrint();
console.log('start');

// start
// 소나타
// SM5
// 그랜저
// Last : 소나타 SM5 그랜저
```


10. 콜백 패턴

동기식 프로그램에서 일련의 연속적 연산 단계로 코드를 실행 하는 블로킹인데 비동기 프로그래밍에서는 이전 작업 실행이 완료 되지 않아도 다음 작업이 실행이 되는데 이전 작업이 완료 되었을 경우 다음 작업이 실행 할 수 있도록 하는 것이 콜백 패턴 이다.

01. 연속전달 방식 (CPS : continuation-processing style)

- 작업이 종료 되었을 때 완료 이후 실행 할 수 있는 함수를 인자로 전달 하는 방식으로 일반적으로 마지막 파라미터에 정의
- 단순히 결과를 호출자에게 직접 반환하는 대신 결과를 다른 함수(콜백)로 전달 하는 것을 의미함.
- 동기 함수는 작업을 완료 할 때까지 블로킹, 비동기 함수는 제어를 즉시 반환하고 결과는 Event Loop의 다음 사이클에서 핸들러로 전달

```
// direct style
function add(a, b) {
  return a + b;
}
```

동기식 전달

```
// cps
function addCps(a, b, callback) {
  callback( a + b );
}

// cps call 함수
function addCpsPrint(result) {
  console.log("addCpsPrint :: ", result);
}

console.log("이전");
addCps(1,2, result=> console.log( `Result: ${result}` ));
addCps(1,2, addCpsPrint);
console.log("이후");

결과 : =====
이전
Result: 3
addCpsPrint :: 3
이후
```

비 연속 전달(Non-CPS)

```
const result = [1,3,7].map(e => e - 1);
console.log(result); // [ 0, 2, 6 ]
```

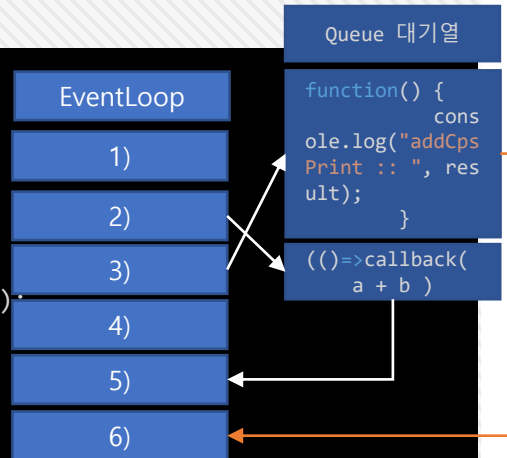
비동기식 전달

```
function addCps(a, b, callback) {
  setTimeout( 5 ) ( )=>callback( a + b ) ,1000);
}

function addCpsPrint(result) {
  setTimeout(
    6) function() {
      console.log("addCpsPrint :: ", result);
    }
    ,1000);
}

1) console.log("이전");
2) addCps(1,2, result=> console.log( `Result: ${result}` ));
3) addCps(1,2, addCpsPrint);
4) console.log("이후");

결과 : =====
이전
이후
Result: 3
addCpsPrint :: 3
```



10. 콜백 패턴

02. Callback hell

- 많은 클로저와 in-place 콜백 정의가 코드의 가독성을 떨어뜨리고 관리할 수 없는 코드 (pyramid of doom)

```
function fprint( a, gCb) {
    a += a;
    gCb(a);
}

fprint(1, function (result) {
    console.log("result = ", result); // result = 2
    setTimeout(() => fprint(result, function (result) {
        console.log("result = ", result); // result = 4
        setTimeout(() => fprint(result, function (result) {
            console.log("result = ", result); // result = 8
            setTimeout(() => fprint(result, function (result) {
                console.log("result = ", result); // result = 16
            },500);
        },500);
    }, 500);
}));
```

- 가장 큰 문제점 : 가독성
 - 함수가 길어 지면 시작과 끝을 알 수 없음
 - 각 스코프에서 사용하는 변수 중복 가능
- 클로저가 성능 및 메모리 소비 측면에서 문제 발생
 - 가비지 수집 시 유지 됨

- 익명 함수로 기명 함수로 변경 하여 가독성 높임
 - 순차적으로 실행 하게 변경

```
result = 2
result = 4
result = 8
result = 16
```

기명 함수 변경

```
function fprint( a, fgCb) {
    a += a;
    fgCb(a);
    return a;
}

const fPrint01 = function(a, fgCb) {
    var result = fprint(a, fgCb);
    setTimeout(fPrint02, 500, result, fgCb );
}

const fPrint02 = function(a, fgCb) {
    var result = fprint(a, fgCb);
    setTimeout(fPrint03, 500, result, fgCb );
}

const fPrint03 = function(a, fgCb) {
    var result = fprint(a, fgCb);
    setTimeout(fPrint04, 500, result, fgCb );
}

const fPrint04 = function(a, fgCb) {
    fprint(a, fgCb);
}

setTimeout(fPrint01, 500, 1, function (result) {
    console.log("result = ",result)
});
```

03. Promise

- 비동기 작업의 최종 결과(또는 에러)를 담고 있는 객체
- 상태 : 대기중(pending:완료 되지 않음), 이행됨(fulfilled:성공적으로 끝남), 거부됨(rejected:에러 종료), 결정됨(settled: 이행되거나 거부됨)
- 이행(fulfillment)값이나 거부(rejection)와 관련된 에러(원인)을 받기 위해 프로미스 인스턴스의 then()함수를 사용 할 수 있음

```
promise.then(onFulfilled, onRejected)
```

- onFulfilled : 프로미스 이행값을 받는 콜백
- onRejected : 거부 이유(에러)를 받는 콜백

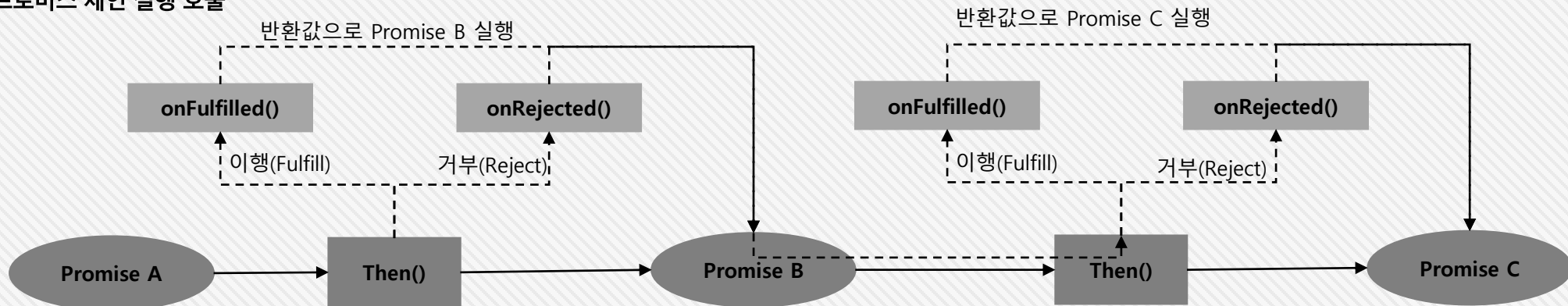
```
asyncOperation(arg, (err, result)=> {  
  if (err) {  
    // 에러 처리  
  }  
  
  // 결과 처리  
})
```

CPS -> Promise

```
asyncOperationPromise(arg)  
  .then(result => {  
    // 결과 처리  
  }, err => {  
    // 에러 처리  
  })
```

```
asyncOperationPromise(arg)  
  .then(result => {  
    // 결과 처리  
    return asyncOperationPromise2(arg2);  
  })  
  .then(result2 => {  
    // 결과 처리  
    return "done";  
  })  
  .then(undefined, err => {  
    // 체인내의 에러를 여기서 catch 함  
  })
```

프로미스 체인 실행 호출



Promise API

- 생성자 : `new Promise((resolve, reject) => {})`
 - `resolve(obj)` : 호출될 때 제공 될 때 이행값으로 Promise를 이행하는 함수. obj가 값 이면 자체 전달, obj가 Promise나 thenable이면 obj의 이행값 전달
 - `reject(err)` : err 사유와 함께 Promise를 거부
- Promise 인스턴스에서 사용 가능한 주요 함수
 - `promise.then(onFulfilled, onRejected)`
 - `promise.catch(onRejected)` : `promise.then(undefined, onRejected)`에 대한 편리한 버전
 - `promise.finally(onFinally)` : `promise`가 결정(이행, 거부) 될 때 호출 되는 콜백, 입력으로 인자는 수신 하지 않으며 여기에서 반환된 값은 무시

```
function accessCall(msg) {
    console.log("성공 : async call ", msg);
}

function failCall(msg) {
    console.log("실패 : reject call", msg);
}

var fPromise = function(param) {

    console.log("비동기 시작");

    return new Promise(function(resolve, reject){
        setTimeout(function() {
            if (param) {
                resolve("ok");
            } else {
                reject("fail");
            }
        }, 100);
    });
}
```

```
console.log("동기 시작");
var rn = fPromise(true)
    .then(function(msg) {
        console.log("성공 : resolve ", msg);
        accessCall(msg);
    })
    .catch(function(msg) {
        console.log("실패 : reject ", msg);
        failCall(msg);
    })
    .finally(function() {
        console.log("비동기 종료 ");
    });

console.log("동기 종료 :: ", rn);
```

```
동기 시작
비동기 시작
동기 종료 :: Promise { <pending> }
성공 : resolve ok
성공 : async call 12 ok
비동기 종료
```

예제 1

```
function delayCall(milliseconds, callBack) {
  console.log("delayCall ... Start .... Sync");
  var p1 = new Promise((resolve, reject) => {
    console.log("delayCall ... Start .... ASync");
    setTimeout(() => {
      console.log("delayCall ... Promise ....resolve(callBack(rn))");
      var rn = {
        "Date": new Date(),
        "callBack": callBack,
        "msg" : "ok"
      }
      resolve(callBack(rn));
    }, milliseconds);
  });

  p1.then((value) => {
    console.log(`delayCall resolve .. End .. ASync .. ${value} .... ${new Date().getSeconds()}`);
  })
  .catch((value)=>{
    console.log(`delayCall reject .. End .. ASync .. ${value} .... ${new Date().getSeconds()}`);
  })
  console.log("delayCall ... End .... Sync");
  return "OK";
}

function fFunc01(param) {
  console.log(`fFunc .... ${new Date().getSeconds()} param ::\n`, param);
  return param.msg;
}

console.log(`##### Start .... ${new Date().getSeconds()}`);
var rnDelayCall = delayCall(1000, fFunc01);
console.log(`##### End .... ${new Date().getSeconds()} .... dcall :: ${rnDelayCall}`);
```

```
##### Start .... 32
delayCall ... Start .... Sync
delayCall ... Start .... ASync
delayCall ... End .... Sync
##### End .... 32 .... dcall :: OK
delayCall ... Promise ....resolve(callBack(rn))
fFunc .... 33 param ::
{
  Date: 2021-08-12T07:31:33.775Z,
  callBack: [Function: fFunc01],
  msg: 'ok'
}
delayCall resolve .. End .. ASync .. ok .... 33
```

- resolve : 함수 호출

Promise 객체의 정적 메소드

- `Promise.resolve(obj)` : 주어진 값으로 이행하는 `Promise.then` 객체를 반환
- `Promise.reject(reason)` : 주어진 이유(reason)로 거부된 `Promise` 객체를 반환

```
console.log("start.....");
Promise.resolve("Success").then(function(value) {
    console.log("Step 1 :: " ,value); // "Success"
}, function(value) {
    // 호출되지 않음
});

var p = Promise.resolve([1,2,3]);
p.then(function(v) {
    console.log("Step 2 :: 배열 :: ", v[0]); // 1
});

var f1 = function fCallback() {
    return "fCallback Success";
}

Promise.resolve(f1()).then(function(value) {
    console.log("Step 3 :: 함수 호출 :: ", value); // "함수 호출 :: fCallback Success"
}, function(value) {
    // 호출되지 않음
});

var p2 = Promise.resolve(p);
p2.then((value)=>{
    console.log("Step 4 :: 다른 Promise 이행 :: ", value); // "다른 Promise 이
행 :: [ 1, 2, 3 ]"
});

console.log("end.....");
```

```
console.log("start.....");

Promise.reject("Testing static reject").then(function(reason) {
    // 호출되지 않음
}, function(reason) {
    console.log("Step 5 :: " , reason); // "Testing static reject"
});

Promise.reject(new Error("fail")).then(function(error) {
    // 호출되지 않음
}, function(error) {
    console.log("Step 6 :: " , error); // Stacktrace
});

console.log("end.....");
```

```
start.....
end.....
Step 1 :: Success
Step 2 :: 배열 :: 1
Step 3 :: 함수 호출 :: fCallback Success
Step 4 :: 다른 Promise 이행 :: [ 1, 2, 3 ]
Step 5 :: Testing static reject
Step 6 :: Error: fail
    at Object.<anonymous> (C:\WPJ\NodeJS\Study\Wbase\promise.js:35:16)
    at Module._compile (internal/modules/cjs/loader.js:1068:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:1097:10)
    at Module.load (internal/modules/cjs/loader.js:933:32)
    at Function.Module._load (internal/modules/cjs/loader.js:774:14)
    at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
    at internal/main/run_main_module.js:17:47
```

Promise 객체의 정적 메소드

- `Promise.race(iterable)` : Promise 객체를 반환합니다. 이 프로미스 객체는 `iterable` 안에 있는 프로미스 중에 가장 먼저 완료된 것의 결과값으로 그대로 이행하거나 거부합니다.

```
var p1 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve('하나'), 500);
});
var p2 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve('둘'), 100);
});

Promise.race([p1, p2])
  .then(function(value) {
    console.log(value); // "둘" 둘 다 이행하지만 p2가 더 빠르므로
  });

var p3 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve('셋'), 100);
});
var p4 = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error('넷')), 500);
});

Promise.race([p3, p4])
  .then(function(value) {
    console.log(value); // "셋" // p3이 더 빠르므로 이행함
  }, function(reason) {
    // 실행되지 않음
  });
```

```
var p5 = new Promise(function(resolve, reject) {
  setTimeout(() => resolve('다섯'), 500);
});
var p6 = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error('여섯')), 100);
});

Promise.race([p5, p6])
  .then(function(value) {
    // 실행되지 않음
  }, function(error) {
    console.log(error.message); // "여섯" p6이 더 빠르므로 거부함
  });
```

Promise 객체의 정적 메소드

- `Promise.all(iterable)` : 순회 가능한 객체에 주어진 모든 프로미스가 이행한 후, 혹은 프로미스가 주어지지 않았을 때 이행하는 Promise를 반환합니다. 주어진 프로미스 중 하나가 거부하는 경우, 첫 번째로 거절한 프로미스의 이유를 사용해 자신도 거부

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values); // [ 3, 42, 'foo' ]
});
```

```
// 매개변수 배열이 빈 것과 동일하게 취급하므로 이행함
var p = Promise.all([1,2,3]);
// 444로 이행하는 프로미스 하나만 제공한 것과 동일하게 취급하므로 이행함
var p2 = Promise.all([1,2,3, Promise.resolve(444)]);
// 555로 거부하는 프로미스 하나만 제공한 것과 동일하게 취급하므로 거부함
var p3 = Promise.all([1,2,3, Promise.reject(555)]);

// setTimeout()을 사용해 스택이 빈 후에 출력할 수 있음
setTimeout(function() {
  console.log(p);      // Promise { [ 1, 2, 3 ] }
  console.log(p2);     // Promise { [ 1, 2, 3, 444 ] }
  console.log(p3);     // Promise { <rejected> 555 }
});
```

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('하나'), 1000);
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('둘'), 2000);
});
var p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('셋'), 3000);
});
var p4 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('넷'), 4000);
});
var p5 = new Promise((resolve, reject) => {
  //setTimeout(() => resolve('다섯'), 4000);
  reject(new Error('거부'));
});

Promise.all([p1, p2, p3, p4])
  .then(values => {
    console.log(values); // [ '하나', '둘', '셋', '넷' ]
  })
  .catch(error => {
    console.log(error.message)
  });
// .catch 사용:
Promise.all([p1, p2, p3, p4, p5])
  .then(values => {
    console.log(values);
  })
  .catch(error => {
    console.log(error.message) // 거부
  });
```


04. Async/await

- ECMAScript 2017 JavaScript 에디션의 일부인 `async functions` 그리고 `await` 키워드는 ECMAScript2017에 추가되었습니다. 이 기능들은 기본적으로 비동기 코드를 쓰고 Promise를 더 읽기 더 쉽도록 줌
- 비동기 함수를 `async` 함수로 만들기 위하여 `function()`앞에 `async` 키워드를 추가합니다. `async function()`은 `await` 키워드가 비동기 코드를 호출할 수 있게 해주는 함수
- `fulfil Promise`가 반환되기 때문에 반환된 값을 사용하기 위해선 `.then()` 블럭 사용
- `await` 키워드는 웹 API를 포함하여 Promise를 반환하는 함수를 호출할 때 사용
- `await`는 `async function` 안에서만 쓸 수 있음

```
function helloSync() { return "Hello" };
console.log(helloSync());                // Hello

async function helloAsync() { return "Hello" };
console.log(helloAsync());                // Promise { 'Hello' }
helloAsync().then((value=>console.log(value))); // Hello
```

```
async function hello01() {
    return greeting = await Promise.resolve("Hello 01");
};

console.log("start.....");
hello01().then(console.log);
console.log("hello01 :: ", hello01, " ", hello01());
console.log("end.....");

결과
start.....
hello01 :: [AsyncFunction: hello01]    Promise { <pending> }
end.....
Hello
Hello 01
```

```
async function asyncError(isParam) {
    try {
        if (isParam) {
            return await Promise.resolve("Hello 01");
        } else {
            throw new Error("ERROR");
        }
    } catch(e) {
        console.log("E ", e); // Stacktrace
        return e.message;
    }
};
```

```
asyncError(true).then((value) => {
    console.log(value); // Hello 01
}).catch((value)=>{
    console.log(value);
});
```

```
asyncError(false).then((value) => {
    console.log(value);
}).catch((value)=>{
    console.log(value); // ERROR
});
```

async/await 성능

```
function timeoutPromise(interval) {
  return new Promise((resolve, reject) => {
    setTimeout(function(){
      resolve("done");
    }, interval);
  });
};

async function timeTest() {
  await timeoutPromise(3000);
  await timeoutPromise(3000);
  await timeoutPromise(3000);
}

async function timeTest01() {
  const timeoutPromise1 = timeoutPromise(3000);
  const timeoutPromise2 = timeoutPromise(3000);
  const timeoutPromise3 = timeoutPromise(3000);

  await timeoutPromise1;
  await timeoutPromise2;
  await timeoutPromise3;
}
```

```
let startTime = Date.now();
timeTest().then(() => {
  let finishTime = Date.now();
  let timeTaken = finishTime - startTime;
  console.log("timeTest :: Time taken in milliseconds: " + timeTaken);
})

// timeTest :: Time taken in milliseconds: 9031
```

```
startTime = Date.now();
timeTest01().then(() => {
  let finishTime = Date.now();
  let timeTaken = finishTime - startTime;
  console.log("timeTest01 :: Time taken in milliseconds: " + timeTaken);
})

// timeTest01 :: Time taken in milliseconds: 3006
```

THANKS