코딩 해본 사람이 빠르게 보는 기초 JS

조교행님

ver.2022

목차

Hello, JS

개발환경세팅

- chocolatey
- vscode
  - Path Intellisense　　파일 위치 자동완성
  - Prettier　　Alt + Shift + F　자동 포매팅
  - VSCode Icons
  - Community Material Theme

JS

- JavaScript 의 줄임말
- 웹에서 쓰이는 유일한 프로그래밍 언어

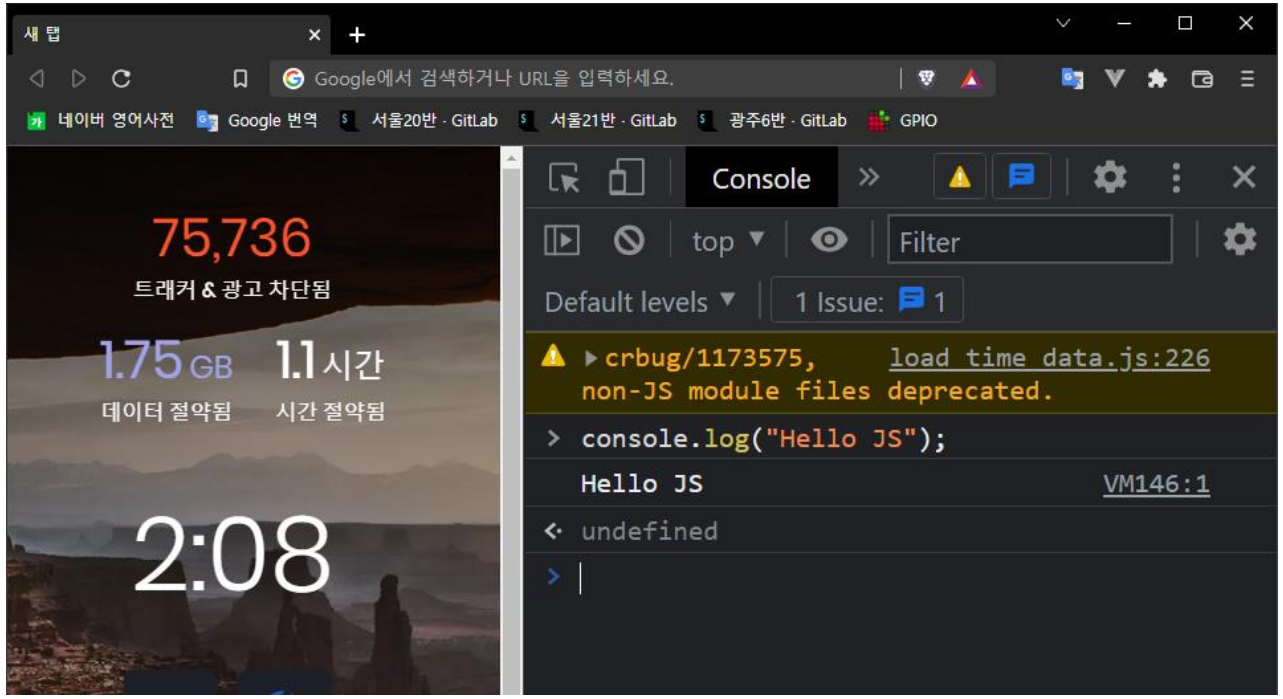
Hello World

JS test.js

C: > Users > mincoding > Desktop > JS test.js

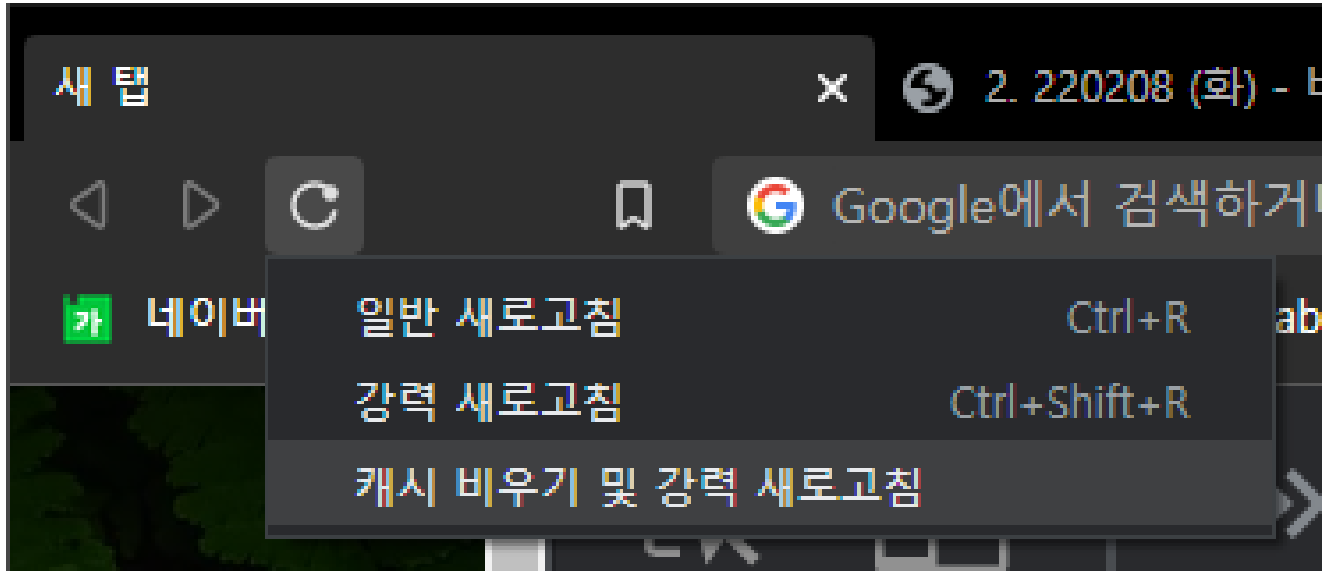
```
1  console.log("Hello JS");
2
```

코드 전체 복사 (Ctrl + A, Ctrl + C)



크롬 키고, 개발자도구 (F12) -> Console
복사한 것 붙여넣고 엔터 -> 실행결과 나옴
왜 작동되는가? 인터넷 브라우저엔 JS 가 깔려있기 때문

캐시 비우기 및 강력 새로고침



개발자도구 켠 상태에서,
새로고침에 오른쪽 버튼 클릭하면 새로고침 종류 선택 가능

코드 실행결과 확인하고 반드시,
캐시 비우기 및 강력 새로고침
눌러줄 것.

console.log()

정확히는, console 객체 안에 log 함수
객체와 함수는 JS 의 알파와 오메가
그런데, 다른 언어에서의 객체, 함수와는 다르다!
즉, 이 문장 하나를 이해하는 게, JS 를 배우는 과정

JS 는 인터프리터 언어다.

- 컴파일 언어 (C, C++, JAVA, C#)
- 인터프리터 언어 (JS, Python)

- 즉, 컴파일이 필요없고 실행기만 있으면 되는데,
- 우리의 실행기가 바로 브라우저다.

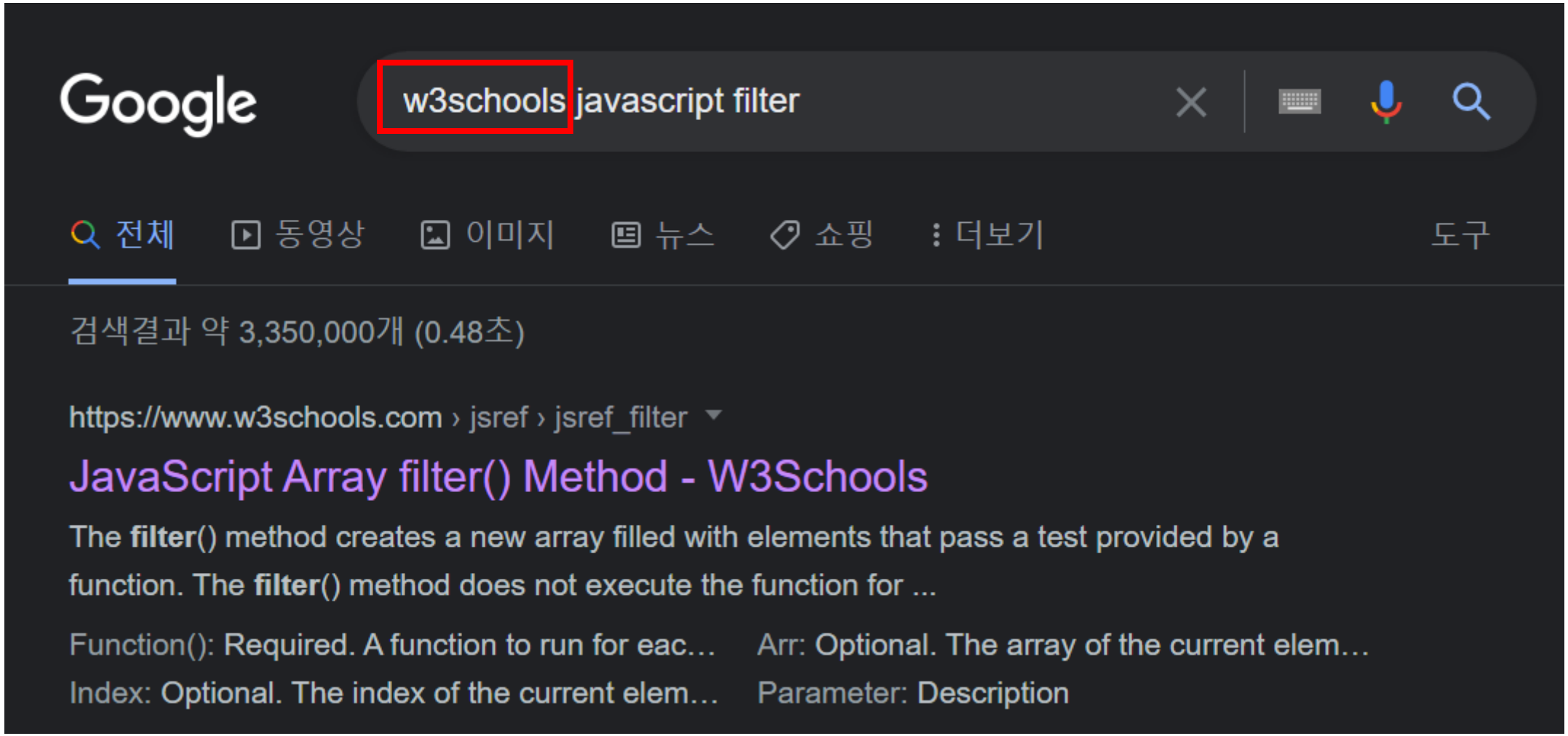
cf) 위의 구분에서도 알 수 있듯,
자바와 자바스크립트는 완전히 다른 언어다.

구글링 팁

구글링 팁

- 웹개발자는 구글링을 생활화해야
- 배울 게 너무 많아서 머릿속에 다 못 넣어둔다

w3schools 전 세계에서 가장 유명한 웹 튜토리얼 사이트



JavaScript Array filter()

❮ Previous

Example 1

Return an array of all values in ages[] that are 18 or over:

```
const ages = [32, 33, 16, 40];
const result = ages.filter(checkAdult);

function checkAdult(age) {
  return age >= 18;
}
```

Try it Yourself »

해당 주제의 개념 설명 전,
가장 일반적으로 활용하는 Example 부터 보여준다.

변수 (variable)

변수를 쓰려면 선언해야

```
const a = 1;
```

- const, let, var

- 90% 의 경우 const 사용
- 10% 의 경우 let 사용
  - 언제? 변수 변경해야 할 때.
    - 대표적으로 for (let i = 0; ;)
    - flag, cnt ...
  - 배열의 엘리먼트, 객체의 프로퍼티 변경은 const 도 가능
- var 는 2015년 이전 유일한 변수선언법이며,
  사용하지 말라(deprecated)

선언은 하되, 타입은 안 붙임

```
int a = 1       (x)
const a = 1     (o)
```

그럼 타입을 어떻게 알 수 있나?

```
console.log(typeof a);
```

자료형 (Type)

자료형 (Type)

- number int, float, double 구분 없이 모두 number
- string "문자열"
  Java처럼 대문자 String 이 아니라 string
  단 하나의 문자도 문자열로 취급. 즉, char 없음.
  작은따옴표로 쓰던 큰따옴표로 쓰던 똑같이 string
  (큰따옴표 권장함)
- boolean true 또는 false
  True 나 False 아님. 소문자로 시작

- array, object, function
  마찬가지로 타입이다. 함수도 타입

```js
const n1 = 3;
const n2 = -4;
const n3 = -0.72;
const s1 = "a";
const s2 = "abc";
const s3 = "3";
const b1 = true;
const b2 = "true";

console.log(n1, typeof n1);
console.log(n2, typeof n2);
console.log(n3, typeof n3);
console.log(s1, typeof s1);
console.log(s2, typeof s2);
console.log(s3, typeof s3);
console.log(b1, typeof b1);
console.log(b2, typeof b2);

// === vs ==
console.log(n1, "==", s3, n1 == s3); // true
console.log(n1, "===", s3, n1 === n3); // false
```

```
3 'number'
-4 'number'
-0.72 'number'
a string
abc string
3 string
true 'boolean'
true string
3 '==' '3' true
3 '===' '3' false
```

3 과 "3" 은 타입이 다르다.

== 두 개. 값 비교 (자동 형변환)
=== 세 개. 타입까지 비교

형변환

```
1  const a = 10;
2  console.log(typeof a);
3  console.log(typeof String(a));
```

```
number
string
```

쓰고자 하는 타입을 대문자로 쓰기

String(), Number() ...

백틱 ``

백틱 ` `

- 출력의 방법 중 하나
- 키보드 왼쪽 위에, 탭키 위에 존재
- 변수 및 함수의 실행 결과를 편하게 출력

```
const myName = "이자룡";
const age = 29;

console.log(`내 이름은 ${myName}, 나이는 ${age}`);
```

```
내 이름은 이자룡, 나이는 29
```

if, while, for

if, while, for

C언어와 완전히 동일하나, 다음을 알아야.

- 조건문에서,

==	(2개)	값만 비교
===	(3개)	타입까지 비교
=!	(1개 + !)	값만 비교
==!	(2개 + !)	타입까지 비교

- 반복문에서,
  - i, j 사용 시 const 대신 let 사용
  - flag, cnt 등, 변해야 하는 변수도 마찬가지

if else

```
const a = 5;
const b = 10;

if (a > b) {
  console.log("a > b");
} else if (a === b) {
  console.log("a === b");
} else {
  console.log("a < b");
}
```

while

```
let i = 1;

while(i <= 10) {
    console.log(i);
    i++;
}
```

for

```
for (let i = 1; i <= 10; i++) {
    console.log(i);
}
```

&&, || 추가 가능

조건에서 false 로 인식하는 것

- false
- 0
- ""
- undefined, null

- 단, 빈 배열 [] 과 빈 객체 {} 는 true

삼항연산자 ? :

삼항연산자 ? :

- if else 조건문을 짧게 쓴 것
- React 에서 자주 씀

```
const myName = "로미오";
const yourName = "줄리엣";

myName === "로미오" || yourName === "줄리엣"
  ? console.log("||실행")
  : console.log("error!");
```

조건

true 일 경우

false 일 경우

배열 (array)

배열 (array)

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

- 중괄호 { } 가 아니라, 대괄호 [ ] 사용
- 크기 선언 불필요
- 다른 타입끼리도 배열 구성 가능. 그런데, 하지 마라.

length

```
const a = "jony";
const b = [1, 2, 3];
console.log(a.length);
console.log(b.length);
```

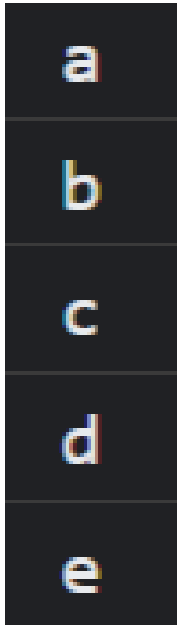
string 문자열의 길이 4

array 배열의 길이 3

a.length() 가 아니라 a.length

index

```
const arr = ['a', 'b', 'c', 'd', 'e'];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```



배열 출력 시 loop 불필요

- 그냥 찍으면 된다

```
const arr = ["a", "b", "c", "d", "e"];
console.log(arr);
```

```
▶ (5) ['a', 'b', 'c', 'd', 'e']
```

객체 (object)

객체 (object)

- 다른 객체지향 언어를 배울때와 접근법이 다르다
- JS 에선 클래스 전에 객체를 배움

- 정의: 키(key) 와 값(value) 으로 구성된 프로퍼티의 모음
- 즉, Python 의 Dictionary

```
const myInfo = {
  name: "조교행님",
  job: "유튜버",
  age: 29,
  isGirlFriend: true,
  familyMember: [
    "아빠",
    "엄마",
    "멍멍이"
  ],
};
```

키(key)　　name, job, age 등 앞에 달려있는 이름
　　따옴표 안 씀. 영어만 허용

값(value)　　"조교행님", "강사", 29 등, 뒤에 달려있는 값

둘을 합쳐서 프로퍼티(속성, property)

프로퍼티는 콤마(,) 로 구분

* 콤마를 빼먹어서 생기는 undefined 가 많음

객체 프로퍼티 접근

```
const myInfo = {
  name: "조교행님",
  job: "유튜버",
  age: 29,
  isGirlFriend: true,
  familyMember: [
    "아빠",
    "엄마",
    "멍멍이"
  ],
};

console.log(myInfo.name);
console.log(myInfo.familyMember);
console.log(myInfo.familyMember[2]);
console.log(myInfo.familyMember[3]);
```

객체 다음에 쩜 . 을 찍어 접근
객체 "안에" 배열의 엘리먼트에 접근 시 당연히 인덱스 접근

```
조교행님
▶ (3) ['아빠', '엄마', '멍멍이']
멍멍이
undefined
```

세번째 인덱스는 정의한 적이 없기 때문에 undefined

undefined

- 웹 개발자가 엄청나게 많이 마주치는 존재
- 대체 왜?
  - 서버에서 보내준 객체에 해당 프로퍼티가 없음
  - 프로퍼티 이름을 잘못 작성해 접근
  - 데이터가 서버로부터 들어오기도 전에 접근 (비동기)
  - ...

undefined vs null

- `undefined` 고의가 아니다. 의도치 않은 결과.
- `null` 고의. 개발자가 의도적으로 비워둔 값.

배열과 객체의 비교는 불가능

```
1  const a = [1, 2, 3];
2  const b = [1, 2, 3];
3
4  console.log(a === b);
```

```
false
```

배열과 객체의 비교는 불가능

```
1  const myInfo1 = {
2      name: "jony",
3      age: 29,
4  };
5  const myInfo2 = {
6      name: "jony",
7      age: 29,
8  };
9
10 console.log(myInfo1 === myInfo2);
```

```
false
```

Parsing

Parsing

- 내가 원하는 형태로 데이터를 바꾸는 것

+ 연산자

- 두 문자열을 합칠 때 사용

```
const a = "jony";
const b = "lee";
console.log(a + b);
```

```
jonylee
```

```
const a = "jony";
const b = 123;
console.log(a + b);
```

```
jony123
```

string 으로 자동 형변환

concat() - 배열

- 두 배열이나 문자열을 합칠 때 사용

```
const arrA = [1, 2, 3];
const arrB = ["jony", "lee"];

const arrC = arrA.concat(arrB);

console.log(arrC);
```

```
▶ (5) [1, 2, 3, 'jony', 'lee']
```

concat() - 문자열

```
const arrA = "jony";
const arrB = "coding";

const arrC = arrA.concat(arrB);

console.log(arrC);
```

```
jonycoding
```

즉, + 연산자와 동일

push, pop

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr, arr.length);
const ret = arr.pop();
console.log(arr);
console.log(ret);
```

const 로 썼음에도 변경 가능
즉, 배열이든 객체든 const 로 선언해도 변경 가능함

```
▶ (4) [1, 2, 3, 4] 4
▶ (3) [1, 2, 3]
4
```

indexOf() - 배열

- 배열 또는 문자열의 인덱스 찾아줌
- indexOf(배열 또는 문자열, 검색 시작 index);
- 못 찾을 경우 -1 리턴

```
const a = [1, 2, 3, 4, 1];
console.log(a.indexOf(1));
console.log(a.indexOf(1, 1));
console.log(a.indexOf(5, 1));
```

```
0
4
-1
```

indexOf() - 문자열

```
const a = "jonyleejonygood";
console.log(a.indexOf("jony"));
console.log(a.indexOf("jony", 2));
console.log(a.indexOf("sylvie"));
```

```
0
7
-1
```

slice()

- slice(시작 index, 끝 index)
  - 시작 index 부터 끝 index 전까지 문자열 또는 배열 얻기

```
const str = "ABCDEFG";
const arr = [1, 2, 3, 4, 5];
console.log(str.slice(1, 4));
console.log(arr.slice(1, 4));
```

```
BCD
▶ (3) [2, 3, 4]
```

replace()

- 특정 문자열을 모두 다른 문자열로 교체

- str.replace(/문자열/g, "바꿀 문자열");

```
const str = "짜장면짬뽕짜장면탕수육";
const result = str.replace(/짜장면/g, "볶음밥");
console.log(result);
```

```
볶음밥짬뽕볶음밥탕수육
```

split()

- 파라미터에 구분 기준을 넣어주어, 배열로 만듦

```
const str = "짜장면 짬뽕 탕수육";
const result = str.split(" ");
console.log(result);
```

이 경우, 공백 " " 이 구분 기준

```
▶ (3) ['짜장면', '짬뽕', '탕수육']
```

split()

```
const str = "짜장면¦짬뽕¦탕수육";
const result = str.split("¦");
console.log(result);
```

이 경우, "¦" 이 구분 기준

```
▶ (3) ['짜장면', '짬뽕', '탕수육']
```

split()

```
const str = "짜장면짬뽕탕수육";
const result = str.split("");
console.log(result);
```

구분 기준이 빈 문자열 ""

```
▶ (8) ['짜', '장', '면', '짬', '뽕', '탕', '수', '육']
```

객체의 변경

- const 로 선언했더라도 변경 가능

```
myInfo.job = "무직";
console.log(myInfo);
```

```
age: 29
familyMember: (3) ['아빠', '엄마', '멍멍이']
isGirlFriend: true
job: "무직"
name: "조교행님"
```

객체의 프로퍼티 추가

- 쩜(.) 으로 추가하는 대신 대괄호 [ ] 사용

```
const myInfo = {
  name: "조교행님",
  job: "유튜버",
  age: 29,
  isGirlFriend: true,
  familyMember: ["아빠", "엄마", "멍멍이"],
};

myInfo["address"] = "경기도 안양시";
console.log(myInfo);
```

```
address: "경기도 안양시"
age: 29
familyMember: (3) ['아빠', '엄마', '멍멍이']
isGirlFriend: true
job: "유튜버"
name: "조교행님"
```

객체의 프로퍼티 추가

- 다음은 의도와는 다른 결과를 만듦

```
const myInfo = {
  name: "조교행님",
  job: "유튜버",
  age: 29,
  isGirlFriend: true,
  familyMember: ["아빠", "엄마", "멍멍이"],
};

const myKey = "address";
const myValue = "경기도 안양시";
myInfo.myKey = myValue;
console.log(myInfo);
```

찜 . 으로 추가한 경우

```
age: 29
familyMember: (3) ['아빠', '엄마', '멍멍이']
isGirlFriend: true
job: "유튜버"
myKey: "경기도 안양시"
name: "조교행님"
```

개발자가 원한 것

address: "경기도 안양시"

객체의 프로퍼티 추가

- 대괄호로 추가하면 문제 해결

```
const myInfo = {
  name: "조교행님",
  job: "유튜버",
  age: 29,
  isGirlFriend: true,
  familyMember: ["아빠", "엄마", "멍멍이"],
};

const myKey = "address";
const myValue = "경기도 안양시";
myInfo[myKey] = myValue;
console.log(myInfo);
```

```
address: "경기도 안양시"
age: 29
familyMember: (3) ['아빠', '엄마', '멍멍이']
isGirlFriend: true
job: "유튜버"
name: "조교행님"
```

대괄호 [ ] 로 추가한 경우

함수 (function)

함수 (function)

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

a = add(5, 3);
console.log(a);
```

```
8
```

시작 시 function 키워드 사용

리턴, 파라미터의 타입 없음

지역변수 a, 전역변수 a 차이는 알고 있을 것

선언 순서는 상관없음

```
function func1(a, b) {
  func2();
  return String(a) + String(b);
}

function func2() {
  console.log("jony");
}

const a = func1(5, 3);
console.log(typeof a);
console.log(a);
```

func2() 가 나중에 선언되었음에도 사용 가능

그러나, 그런 식으로 쓰지 마시오...

JS 에선, 함수도 타입이다

```
const go = function () {
    console.log("#");
};

go();
```

function 의 실행 결과가 아니라,
함수 자체가 들어감

그래서, go 라는 함수를 실행시킬 수 있는 것

화살표함수를 배우면 익숙해짐

여기서, go 에 들어가기 전까지는
함수에 "이름이 없음"

이렇게, 이름 없는 함수를 람다식(lambda expression)
이라고 하는데, 한국어로 무명함수

```
const go1 = function () {
    console.log("#");
};

const go2 = go1;
go2();
console.log(typeof go2);
```

go1 에는 소괄호 () 쓰지 않음

소괄호는 함수를 실행시킨다는 뜻이기 때문

즉, go2 에는 함수의 실행결과가 아닌 함수 자체가 들어감

만약 소괄호를 써버리면?

```
const go1 = function () {
    console.log("#");
};

const go2 = go1();
go2();
console.log(typeof go2);
```

```
undefined
▶Uncaught TypeError: go2 is not a function
    at <anonymous>:7:1
```

go2 는 function 이 아니라 undefined

화살표 함수 (arrow function)

- function 키워드 이외에 또 다른 함수 선언법

```
const add = (a, b) => {
   return a + b;
};

a = add(5, 3);
console.log(a);
```

add 라는 변수 안에, 함수 자체가 들어간 것

파라미터 다음에 화살표(arrow),
그 다음 중괄호로 함수의 내용

function 키워드로 만드는 법,
arrow function 으로 만드는 법
둘 다 많이 쓰임

모든 화살표함수는 람다식

```
8
```

화살표 함수 (arrow function)

- 다음과 같이 생략해서 표현 가능

```
const add = (a, b) => a + b;


a = add(5, 3);
console.log(a);
```

```
8
```

함수의 내용이 리턴 한 줄일 때,
중괄호 생략, 리턴 키워드 생략

만약 리턴 없는 함수인데 한 줄이면
중괄호 생략 가능

화살표 함수 (arrow function)

- 만약 리턴 없는 함수인데 한 줄일 경우도 중괄호 생략 가능

```
const add = (a, b) => console.log(a + b);
add(5, 3);
```

```
8
```

console.log()

- console 객체 안에
- log() 함수

- 즉, 객체의 프로퍼티는 함수도 가능하다.
- 값으로 함수가 들어간다? "메서드(method)"

배열 순회 메서드

배열 순회 메서드

- 배열을 다룰 때 자주 쓰는 메서드들을 배워보며,
- 콜백함수의 사용법을 익힌다

- 콜백은 JS 개발자들이 일상적으로 사용하는 개념이므로,
- 반드시 익혀둬야

forEach()

- 배열의 엘리먼트 갯수만큼 콜백함수 호출
- 콜백함수 호출될 때마다,
- 엘리먼트 하나씩 파라미터로 들어감
- 리턴 없음

```
const test = (data) => console.log(data);

const datas = [3, 5, 4, 2];
datas.forEach(test);
```

```
3
5
4
2
```

총 4번 test 함수 호출되고,

호출될때마다 data 로
3, 5, 4, 2 들어감

콜백 (callback)

- 함수 안에 파라미터로 들어가는 함수 "선언문"
- 당장 실행 안되고, 특정한 조건을 만족시켜야만 실행됨

```
const test = (data) => console.log(data);

const datas = [3, 5, 4, 2];
datas.forEach(test);
```

test() 라고 쓰지 않고 test 라고 썼다.
즉, 함수 실행 결과가 파라미터로 들어가는 게 아니라
함수 선언 자체가 들어간 것

파라미터 안에 직접 선언 가능

```
const datas = [3, 5, 4, 2];
datas.forEach((data) => console.log(data));
```

왜 굳이 이렇게?
한번 쓰고 말 함수를 굳이 이름붙여가면서 선언할 이유가 없음

파라미터로 함수 선언을 바로 넣을 땐, 반드시 람다식이어야
(const 함수이름 =) 이 부분 없음

굉장히 많이 사용하는 기법

참고. 복수/단수 기법

- 배열은 복수로 -s 이름짓고, (datas)
- 각각의 엘리먼트는 단수로 받는다 (data)
- 코드를 직관적으로 이해하기 훨씬 편해짐
  - ex) items/item, infos/info, numbers/number

```
const datas = [3, 5, 4, 2];
datas.forEach((data) => console.log(data));
```

some()

- 리턴에 조건을 달아서, 하나라도 맞으면 true
- 즉, 검증 시 사용

```
const datas = [3, 5, 4, 2];
const isData = datas.some((data) => data > 4);
console.log(isData);
```

4 보다 큰 게 하나라도 있는가?

```
true
```

some()

- 이 경우는 하나도 해당사항 없으니 false

```
const datas = [3, 5, 4, 2];
const isData = datas.some((data) => data < 0);
console.log(isData);
```

every()

- some() 과는 반대로, 모든 엘리먼트가 조건을 만족해야 true

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.every((data) => data > 4));
```

false 반환

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.every((data) => data > 0));
```

true 반환

find()

- 조건을 만족하는 첫번째 값 반환

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.find((data) => data > 3));
```

결과: 4

5도 있지만, 첫번째 값만 반환

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.find((data) => data > 5));
```

undefined

findIndex()

- 조건을 만족하는 첫번째 요소의 "인덱스" 반환

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.findIndex((data) => data > 3));
```

3 리턴

```
const datas = [1, 2, 3, 4, 5];
console.log(datas.findIndex((data) => data > 5));
```

-1 리턴(없는 경우)

map()

- forEach() 와 비슷하나, 리턴값 있음
- 결과를 모아 새로운 배열을 리턴
- React 에서 매우 많이 사용

```
const datas = [3, 5, 4, 2];
const newDatas = datas.map((data) => data + 1);
console.log(newDatas);
```

```
▶ (4) [4, 6, 5, 3]
```

map()

- index 도 활용 가능

```
const datas = [3, 5, 4, 2];
const newDatas = datas.map((data, idx) => {
  return `${idx}: ${data + 1}`;
});
console.log(newDatas);
```

```
▶ (4) ['0: 4', '1: 6', '2: 5', '3: 3']
```

map()

- 덧셈하는 예
- map() 은 새 배열을 리턴하지만, 여기선 리턴을 받지 않음
- map() 을 forEach() 로 바꿔도 똑같은 결과

```
let sum = 0;
const datas = [3, 5, 4, 2];
datas.map((data) => (sum += data));
console.log(sum);
```

참고: for loop 보다 나은 점

- 지금까지 배운 배열 순회 메서드는
- for 문으로 대부분 대체 가능
- 대체 어떤 장점이 있길래 for 문을 안쓰나?

1. 원래 배열을 변경할 위험 줄어듬 (immutability)
    - 웹개발자들은 데이터의 변경을 매우 싫어한다.
2. 가독성 향상
    - 시작조건, 반복조건, 종료조건 생각 안해도 됨

단, 성능상 for loop 와 배열 순회 메서드는 별 차이가 없음

filter()

- 조건을 만족하는 값들로 새로운 배열을 만듦
- React 에서 매우 많이 사용

```
const datas = [3, 5, 4, 2];
const newDatas = datas.filter((data) => data > 2);
console.log(newDatas);
```

```
▶ (3) [3, 5, 4]
```

filter()

- 마찬가지로, index 활용 가능

```
const datas = [3, 5, 4, 2];
const newDatas = datas.filter((data, idx) => idx > 1);
console.log(newDatas);
```

```
▶ (2) [4, 2]
```

배열의 인덱스가 1보다 큰 것만 새 배열로 만듦

기타, 중요한 기능

sort()

- JS 의 정렬은 직관적이지 않음

```
const numbers = [40, 100, 1, 5, 25, 10];
numbers.sort();
console.log(numbers);
```

```
▶ (6) [1, 10, 100, 25, 40, 5]
```

- 기본적으로 문자열 정렬이다.
- sort() 파라미터로, 정렬 기준이 되는 "콜백함수" 를 삽입

오름차순(ascending order)

```
const numbers = [40, 100, 1, 5, 25, 10];
numbers.sort((a, b) => a - b);
console.log(numbers);
```

```
▶ (6) [1, 5, 10, 25, 40, 100]
```

내림차순(descending order)

```
const numbers = [40, 100, 1, 5, 25, 10];
numbers.sort((a, b) => b - a);
console.log(numbers);
```

```
▶ (6) [100, 40, 25, 10, 5, 1]
```

정렬은 제발 외우지 말고,
이 페이지를 캡쳐하던지
구글링해서 필요할때마다 쓰는 걸 추천

setTimeout()

- 콜백함수와 밀리초를 미리 배정해두고,
- 해당 시간 "후에" 실행

```
setTimeout(() => {
  console.log("hi");
}, 1000);
```

```
hi
```

1000 밀리초, 즉 1초 뒤!

setInterval()

- 콜백함수와 밀리초를 미리 배정해두고,
- 해당 시간 "마다" 실행

```
setInterval(() => {
  console.log("hi");
}, 1000);
```

```
4 hi
```

현재까지 동일한 내용이 4번 찍혔고, 앞으로도 계속 찍힐 것.

Date 객체

- 날짜 및 시간 출력 시 사용

```
const date = new Date();
console.log(date);
```

```
Wed May 18 2022 11:19:45 GMT+0900 (한국 표준시)
```

- 당연히, 보기에 불편하다.

Date 객체

```
const date = new Date();
console.log(date.getFullYear());
console.log(date.getMonth() + 1);
console.log(date.getDate());
console.log(date.getHours());
console.log(date.getMinutes());
console.log(date.getSeconds());
```

```
2022
5
18
11
24
48
```

주의사항.
1. getYear 가 아닌 getFullYear
2. getMonth() + 1　　왜? 0부터 11 까지거든.

수고하셨습니다.