

CodeIt

Ctrl + Shift + N : 새로운 노션 창

Cmd/Ctrl + Option/Alt + T : 모든 토글 목록을 펼치거나 닫습니다.

- 영화앱 처음부터 비슷하게 만들어보려고 혼자 쳐보고, 안되면 내가 했던 코드들 참고하면서 무튼 비슷하게 완성, css 내 스타일대로 만들어보고 내가 추가하고자 하는 것들 해서 포폴 하나 완성시키기 겨울방학동안

▼ 목차

- 프로그래밍 오버뷰 16
- 자바스크립트 초급 20
- html/css 21
- 자바스크립트 중급 ~4
- 유닉스 커맨드 26
- 리액트 2
- 백엔드 6
 - node
 - express
- SQL 데이터베이스 10
- Git 11

부스트코스 http, 프론트&백 기본

▼ 프로그래밍 오버뷰

- 서버, 클라, db
 - 서버
 - ex) 클라이언트가 좋아요 누르면 좋아요를 db에 넣어줌, 클라이언트가 복잡한 연산을 할 때 클라 컴에서 하기 버거운 경우 서버가 대신 해주는, 서버가 스케줄에 맞게 작업을 예약하는 경우 (정기 결제) 역할 등
 - 클라이언트 - 이 코드도 내 컴에 있는게 아닌 ip 주소로 요청하면 서버가 주는 것, 그걸 브라우저가 해석해서 띄워주는 것임
 - html
 - css
 - js - 재생버튼, 자막버튼 시 누르면 변화되는 과정
 - 데이터베이스
 - 서버안에 존재하는 것 (유저 모두가 공유하는)
 - 구현방법



spotify 세부 구현 내용 예시

- 라이브러리, 프레임워크 - 남이 써둔 코드
- 분야
 - 웹
 - 컴, 폰에서 모두 가능
 - 아직 수요 가장 큼
 - 복잡해질수록
 - 리액트, 앵귤러, 뷰
 - api 개발 - 두 프로그램 소통하게 해주는 것, 클라와 서버가 소통하게 해주는 역할
 - 자바 피이썬 코틀린 php 루비 js
 - 데이터베이스
 - sql - 규칙엄격, 안정적
 - nosql - 유연 간편 - 프론트면 이걸로 가볍게 경험 ㄱ
 - 모바일
 - 클라이언트 = 애플리케이션
 - 애플 - 스위프트
 - 안드로이드 - 코틀린
 - 따로 만든다는 것이 까다로움
 - 크로스 플랫폼 → 리액트 네이티브 (리액트를 발전시켜서 나온 것임, 웹의 것을 재활용)
 - 플러터 (구글) - dart라는 언어임
 - 베스트?
 - 스위프트 코틀린이 모바일 최적화
 - 리액트네이티브 플러터 - 높은 성능 필요x 시, 개발 비용 적음,
 - 리액트 - js 리액트 안다면
 - 플러터 - 필요한 것만 배워서 빠르게 하고싶다면

- 데이터
 - 데이터 엔지니어
 - 많은 양의 데이터 효과적 처리
 - db 백데이터 특화
 - 백엔드가 대체하는 경우도.
 - 대형 회사에서 존재, 준비단계 해주는 거임
 - 데이터 애널리스트
 - 활용해서 직관적분석
 - sql로 데이터 추출, 파이썬 분석
 - 인사이트 팀원들에게 전달
 - 데이터 사이언티스트
 - 머신러닝 이용, 미래 예측
 - 파이썬
 - 머신 러닝을 서비스에 도입 시킬 방법 고민
 - 머려 엔지니어
 - 백엔드가 맡기도함
 - 머려 알고리즘을 서비스에 녹여내는
 - 머신러닝 리서처
 - 새 머려 알고리즘 찾고 기존 알고리즘 개선
 - 수학, 통계 등 석박사 학위
- 게임
 - 유니티, 언리얼엔진
- 블록체인
 - 비트코인, 쓸라나 등
 - 논란이 많음
 - 탈중앙화
 - 블록체인 엔지니어 - 블록체인 위 올라가는 앱을 만드는
 - 이더리움, 솔리디티 배워야
- 분야 고민
 - 일단 웹 or 데이터

06 - 데이터, 백엔드, 프론트에 대한 전반적인 가이드!

- 요즘 파이썬 - 비개발자여도 영향
 - 이젠 데이터를 기반으로 의사결정하는 시대! - 파이썬 이용
 - 업무 자동화 - 엑셀 취합, 정리, 전송
 - 협업
 - pm - 컵 전공 이후 2번째로 가장 많이 하는 직업

- 어떻게 해야 좋은 코드?
- 객체지향 (기능과 데이터를 하나로 묶은 작은 단위)
 - 데이터나 기능에 변화가 생겨도 관련 객체만 변경
 - 더 의미있는 단위로 나누기 때문에 이해 수월, 재사용 수월
 - 전체가 완성되지 않아도 일부 기능만 사용가능 - 테스트 유리

하지만 잘 나누고 설계 어려우므로 혼자 바로 만들 수 있거나 데이터 소규모 등은 top down으로 빠른 과정, 빠른 컴파일 속도의 것 만들면 됨



▼ 자바스크립트 유튜브 내용 핵심

변수를 선언할때는

- 변하지 않는 값은 const
- 변할 수 있는 값은 let으로 선언

```
const message3 = `My name is ${name}`;
const message4 = `나는 ${30+1}살입니다.`;

console.log(typeof 3);
```

숫자 + 문자열 = 문자열로 변환된다

alert(prompt()) - 창으로 알려준다, 메세지를 보여준다

prompt - 입력받을때 사용

confirm - 사용자에게 확인이나 취소 확인할때

String() → 문자형으로 변환

Number() → 숫자형으로 변환

Number("1234"), 1234

Number("124adsf"), nan

Number(true), 1

Boolean(0),

Boolean(""),

Boolean(null),

Boolean(undefined),

Boolean(NaN), - 빼고 전부 true

String(3),

String(true),

String(null), - 그대로 찍힘 “null”

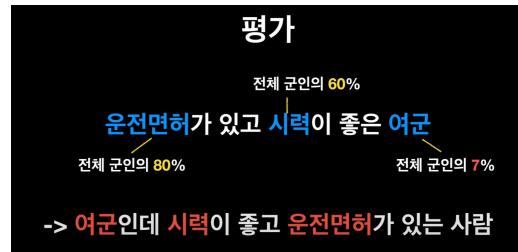
- 어떤값이 들어와도 5 이상 들어오면 안될때 %5

숫자형 문자형 비교했을때 같게 나올때도 있다

- === 할때 type까지 비교한다

반복문에서는

```
for(let i =0;i<10;i++)
```



```
function sayHello(name = 'friend'){
let msg = `Hello, ${name}`
```

```
function sayHello(name){
  console.log(error)
}

let sayHello = function (name) {
}

let sayHello = () => {
```

▼ 함수 표현식

함수선언문 : 어디서든 호출 가능

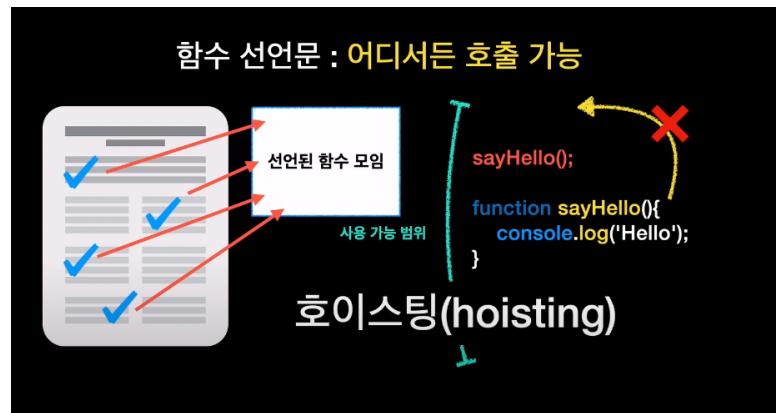
- 아래로 내려가면서 읽는 언어인 인터프리터이다



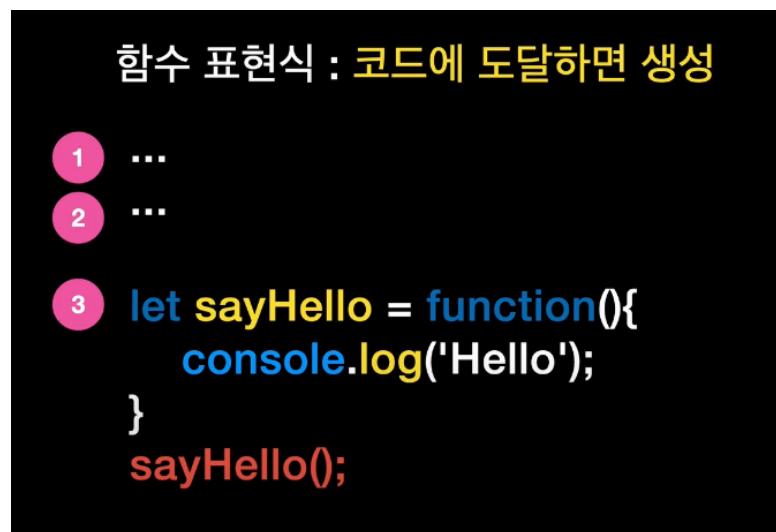
어떻게 코드가 실행될수 있었을까?

- 호이스팅

초기 모든 함수 선언문 찾아서 모임을 만든다



함수 표현식은 코드에 도달하면 생성이 된다



뭐가 더 좋을까?

함수 선언문 쓰는게 더 자유롭다

화살표 함수

```
//함수 표현식  
showError();  
  
let showError = function(){  
    console.log('error');  
}  
  
함수 표현식이기 때문에 실행이 안됨  
-----  
//함수 선언문  
showError();  
  
function showError(){  
    console.log('error');  
}  
-----  
//화살표 함수  
let showError = () =>{
```

```
    console.log('error');
}

showError();
```

```
const sayHello = function(name){
  const msg = `Hello, ${name}`;
  console.log(msg);
};

sayHello("euisung");

//화살표함수
const sayHello = (name) => {
  const msg = `Hello, ${name}`;
  console.log(msg);
};

sayHello("euisung");

//function을 지우고 매개변수 뒤에 화살표를 그려주면 끝
```

```
const add = function(num1,num2){
  const result = num1 +num2 ;
  return result;
};

console.log(add(1,2));

const add = (num1,num2) =>{
  return result = num1 +num2 ;
};

console.log(add(1,2));

const add = (num1,num2) =>(num1 +num2) ;
console.log(add(1,2));
```

- 객체

```
const superman = {
  name : 'clark',
  age : 30,
}

//추가, 접근도 이렇게 2가지 방식으로.
superman.hairColor = 'black';
superman['hobby'] = 'football';
//삭제
delete superman.age;
```

```
function makeObject(name,age){
  return {
    name : name,
    age : age,
    hobby : 'football'
  }
}

const Mike = makeObject('Mike',30);
```

```
console.log('birthday' in Mike); //false
-----
//key가 없으면 무조건 true 나오니까 조건 추가 해주는 예시
function isAdult(user){
```

```

if(!('age' in user) || user.age < 20){
    return false;
}
return true;
}

//a in Mike 면 Mike의 모든 key가 a에 들어감
for(key in Mike){
    console.log(Mike[key]) //value 값이 나옴
}

```

- 객체 내 method

```

let boy = {
    name: "Mike",
    showName : function(){
        console.log(boy.name)
        console.log(this.name) //boy보단 이게 좋음
        console.log('hello my name is ${this.name}') //동일한듯
    }
};

boy.showName();

let man = boy;
man.name = "tom"
man.showName()
    //이 때를 위해 boy.name으로 한정짓는 게 아니라 this.name이 좋음

```

→ 객체 만들때

- this를 사용하고
- 화살표 메소드는 사용하지 말자
 - 화살표 함수는 일반 함수와는 달리 자신만의 this를 갖지 않는다

- 배열

```

let days = ["mon", "tue", "wed"];
days[1]='화요일'

days.push('thus');
days.unshift("sun");
console.log(days)

["sun", "mon", "화요일", "wed", "thus"]

//shift 와 unshift는 첫번째 배열요소 접근

```

```

for(let index = 0; index<days.length; index++)
{
    console.log(days[index]);
}

for(let day of days)
{
    console.log(day);
}

```

- of를 사용

- fetch

`fetch()` 함수는 첫번째 인자로 URL, 두번째 인자로 옵션 객체를 받고, Promise 타입의 객체를 반환합니다. 반환된 객체는, API 호출이 성공했을 경우에는 응답(response) 객체를 resolve하고, 실패했을 경우에는 예외(error) 객체를 reject합니다.

```

fetch(url, options)
    .then((response) => console.log("response:", response))
    .catch((error) => console.log("error:", error));

```

▼ 자바스크립트 초급1

- 추상화
 - 원활한 소통방식, 꼭 필요한 핵심, 구체적 정보 숨기고
 - 지도
- $5^{**}2$ 는 거듭제곱 = 25
거듭제곱이 곱셈보다 우선순위 높음
- 백틱` 의 기능
`이 안에 여러 '' 가 존재할 때 \를 앞에 쓰는 대신 백틱으로 감싸서 가독성을 높이는 방법도 있다`
- typeof
모든 연산보다 우선, string 리턴
- NaN
숫자임, Boolean() 안에 넣으면 false
- 비교연산
 - 자동 형변환이 되기도 함
 - nan은 어떤거랑 연산해도 nan 나옴
 - 관계 비교 시 비교 불가하면 false 나옴 < >
 - === 는 형변환 안됨, ==는 됨
====는 형까지 비교하게됨 안전, 값만 비교하려면 == 쓰기
- 템플릿 문자열
 - console.log(` \${name} 님의 근무 시간은 총 \${time}시간이며, 최종 급여는 \${total}원 입니다.`);
- null undefined - 값이 없다는 의미는 동일
 - undefined 선언해준 다음 값 할당 안했을 때 - 사용자한테 알려주는 느낌
 - null 의도적으로 비어있는 것을 지정할 때 - 사용자가 일부러 넣어놓은 느낌
 - == 이긴 하지만 ===는 false
- x += 1, x++ 가능
- return 없는 함수를 콘솔log 찍으면 undefined
console.log(prnts(3))
- 파라미터 있는 함수에서 파라미터 없이 call 하면 unde
함수 선언 시 초기값 넣으면 됨, 하지만 이 파라미터는 제일 뒤에 오도록!
orderSetMenu(sandwich, drink='스프라이트')
- 같은 변수면 지역변수 먼저 쓰고 없으면 글로벌 변수
- const 상수
 - 변경하려고 하거나 선언 시 값 할당 안해주면 에러
- switch

▼ 예제

```
// 각 등급별 가격
let VIPPrice = 15;
let RPrice = 13;
let SPrice = 10;
let APrice = 8;

// 각 등급에 맞는 가격을 출력하는 함수 checkPrice를 완성하세요
function checkPrice(grade) {
    let price
    let flag=0
```

```

switch(grade){
    case 'VIP':
        price = VIPPrice;
        break;
    case 'R':
        price = RPrice;
        break;
    case 'S':
        price = SPrice;
        break;
    case 'A':
        price = APrice;
        break;
    default:
        flag=1
        console.log("VIP, R, S, A 중에서 하나를 선택해 주세요 .")
}
if(flag==0) console.log(` ${grade}석은 ${price}만원 입니다. `)

// 테스트 코드
checkPrice('R');
checkPrice('VIP');
checkPrice('S');
checkPrice('A');
checkPrice('B');

```

- 피보나치

▼ 자바스크립트 초급2 (자료형)

객체

- 자바스크립트의 모든 것이 객체다!
- 하나의 속성 property = key+value
- key는 따옴표 생략해도됨
 - 그러면 띄어쓰기 금지, - 금지, 문자로만 시작 (\$,_ 가능)
- value는 뭐든 가능
- 접근 시 .표기법으로 안될 경우도 있으니
codeit["born year"] 이렇게 접근하자
- 객체 안 객체 - 계속 이어서 하면됨
- 없는 프로퍼티 접근 시 undefi
 - 수정은 덮어쓰기
 - 추가는 없는 키값에 그냥 할당해주기
 - 삭제는 (delete 프로퍼티) delete myVoca.local;
 - 키값 확인은 (키 in 객체)로 하자
- 객체 내 메소드
 - let showError = function(){ console.log('error'); }
 - showError : function() {},
함수 표현식과 비슷
 - 대부분 . 표현법 쓰긴함 console.log('d') 대표적
 - 다른 객체에서 각 같은 메소드 이름 사용 가능

```

let myVoca = {
  // 각 메소드들도 하나의 프로퍼티므로 ,구분!
  addVoca : function(word, meaning){
    myVoca[word] = meaning
    // .word면 word라는 key를 찾는 것이 됨, 변수일 땐 []로 접근! (함수 케이스)
  }
}

```

```

},
deleteVoca : function(word){
    delete myVoca[word]
},
printVoca : function(word){
    console.log(`"${word}"의 뜻은 "${myVoca[word]}"입니다.`)
}
};

// addVoca메소드 테스트 코드
myVoca.addVoca('parameter', '매개 변수');
myVoca.addVoca('element', '요소');
myVoca.addVoca('property', '속성');
console.log(myVoca);

// deleteVoca메소드 테스트 코드
myVoca.deleteVoca('parameter');
myVoca.deleteVoca('element');
console.log(myVoca);

// printVoca메소드 테스트 코드
myVoca.printVoca('property');

-----
let myVoca = {
    function: '함수',
    constant: '상수',
};

//둘 중에 하나의 방식대로
delete myVoca.constant
delete myVoca['constant']

```

`myVoca[key] → myVoca['parameter']` 되지만

`myVoca.key → myVoca.'parameter'` 안되는듯, key로 접근되는듯

- for in에서 객체의 프로퍼티는 어떤 순서로 정렬이 되는 걸까요!?
 - 숫자 형태 key는 오름차순, 나머지는 추가된 순서대로.
- 프로퍼티 key는 num 가능하지만 실 typeof에선 string으로 쓰임
 - myObject['300'] 처럼 [] 방법으로만 접근 가능
- 내장객체 Date
 - let date1 = new Date('YYYY-MM-DDThh:mm:ss')
 - (yyyy,mm,dd,hours, ~) 도 가능
 - () 면 이 객체 생성 순간
 - month는 0부터 시작, 4는 5월
 - date1.getTime() = timestamp = 700101 기준 얼마나 시간 지났는지
 - date1.getDate()는 일자
 - date1.getDay()는 요일, 4면 목요일
 - date1.toLocaleString() 은 사용자 브라우저 설정 국가 표기 맞춰 날짜와 시간
 - date1.set~()으로 설정도 가능
 - Date.now()는 객체 생성 안해도 현 시점 타임스탬프 확인 가능
 - let timeDiff = myDate2 - myDate1;
 - 객체끼리 사칙연산 가능 (둘의 .getTime()으로 연산)

배열

- 객체로 만들어보니 프로퍼티 name보다 값들의 순서가 중요할 때?
- index = 프로퍼티 name = 0,1,2, / array[1]
- 배열도 객체처럼 key가 인덱스가 되어 for key in 가능!
- array.length

- 추가 수정 삭제
 - 추가 - 없는 인덱스에 값 할당
 - 수정 - 덮어씌우기 값 할당
 - 삭제 - delete array[1] 하면 값만 삭제되니 다른방법!
 - array.splice(1) 하지만 이후 존재하던 234도 사라짐
 - (1,2) 1번 인덱스부터 2개를 삭제
 - (1,2,삭제된 부분에 추가할 요소 'a', 'b') 더 많이 넣어주면 뒤에것은 밀림 = 수정, 그냥 추가도 가능(두번째에 0)

```
arr.splice(삭제할index, 삭제할개수, 추가할요소, 추가할. )
```

▼ 주의 코드

```
for(let i=0;i<numbers.length;i++){
  if(numbers[i]%2==1){
    numbers.splice(i,1)
    i-- //삭제된 자리에 다음까가 올테니 그자리 한번 더 검사
  }
}
```

- shift pop unshift push

```
// 배열의 첫 요소를 삭제: shift()
members.shift();
console.log(members);
// 배열의 마지막 요소를 삭제: pop()
members.pop();
console.log(members);
// 배열의 첫 요소로 값 추가: unshift(value)
members.unshift('NiceCodeit');
console.log(members);
// 배열의 마지막 요소로 값 추가: push(value)
members.push('HiCodeit');
console.log(members);
```

shift는 배열 모두 index 밀리므로 0번째 쪽 관련

- 배열 다른 메소드

```
let brands = ['Google', 'Kakao', 'Naver', 'Kakao'];
console.log(brands.indexOf('Kakao')) //첫번째 1, 없으면 -1
console.log(brands.lastIndexOf('Kakao')) //마지막 3
console.log(brands.includes('Kakao')) //true
brands.reverse();
```

- 배열 for of = 인덱스 활용안하고 모든 값 그냥 추출

```
let influencer = ['suwonlog', 'small.tiger',
'Minam.ludens', 'cu_convenience24']

for (let i = 0; i < influencer.length; i++) {
  console.log(influencer[i]);
}

for (let element of influencer) {
  console.log(element);
}

for (let index in influencer) [
  console.log(influencer[index]);
]
```

다 같은 결과긴 하지만 배열은 for of가 최적화

- 다차원 배열

```
let a = [ [1,2], [3,4] ]
```

자료형 심화

- 5.3e3=5300
-9.1e-5=-0.000091
- 0xff - 16진수
0o377 - 8진수
0b11111111 - 2진수 모두 console() 넣으면 255 십진수로
- toFixed(3) 소수점 3째자리까지 반올림
문자열이 됨
앞에 + 붙이면 숫자됨 (Number(~)와 동일)
- toString(2) - 2진수로 변환, 문자열이 됨
255..toString(8) or (255).toString(16)
- Math.~
- interest = +interest.toFixed()
- 스트링도 배열처럼 딴거 다되고 indexOf(), 추가적으로 아래 가능
 - trim()
 - toUpperCase()
 - charAt(3) - 문자 하나
 - slice(시작, 끝) - 부분문자열
- String vs 배열
 - type = string vs obj
 - ===, == all false
 - 문자열은 immutable, 한번 할당 후 수정 불가
- 기본형 참조형
 - 기본형 - num, str 등은 (변수=값) 값이 그대로 저장
 - 참조형 - 객체, 배열 - (변수 = 주소값)
x = 객체, y = x, 이렇게 되면 값이 복사된게 아니라 주소값이 공유된 것이므로 y에서 값 수정해도 x에서도 값이 반영됨
- 객체에서 각 프로퍼티 따오기

```
let x = {  
    numbers: [1, 2, 3, 4],  
    title: 'Codeit',  
};  
let y = x.numbers;  
let z = x.title;
```

- 참조형 ↳ 값만 복사하고싶으면
 - 배열 - slice()
메소드를 호출할 때 파라미터로 아무런 값도 전달하지 않을 경우에 배열 전체를 그대로 리턴
 - 객체 - Object.assign , or, for in문으로 복사해오기 - 함수로 만들던데..?
- const
 - let은 재할당 가능, 이건 불가
 - 결국 코드 돌아가는 동안은 변하지 않기 때문에 변수로 잘 쓰임
 - 짠 상수랑은 어떻게 구분? - 이름 myName⇒MY_NAME

- `const` 로 변수를 선언하게 되면 값을 재할당할 수 없지만, 할당된 값이 객체나 배열일 경우 메소드를 통해서 그 값을 변경 가능
 - `splice`로
- `var`의 치명적 단점
 - `let const`는 블록이든 함수이든 스코프 내에서 선언되면 외부에선 불가하지만 `var`는 블록(`if`, `for`) 등에서 선언해도 전역으로 됨..
 - 중복 선언됨 `var a=1 var a=2`
 - 호이스팅=변수 선언이 끌려 올라가서 줄이 바뀐것처럼 동작
 - +) 마지막 `for`문까지 돌았음을 나타내기 위한 예시 - `return`

```
for(let i=word.length-1;i>=0;i--)
  word2[i]=word[word.length-i-1]
```

▼ HTML CSS

- <!DOCTYPE HTML> html 버전 알려줌
- <i> strong em
- <meta charset="utf-8"> - 브라우저에게 한글 인코딩 할 수 있도록 해줌
- css에서 p 안에있는 i 태그에만 적용하고 싶으면 `p i { }`
- <html> head body
- 링크
 - 누르면 네이버감
- 이미지
 - %로 전체 화면 대 비율로 가능
- 클래스~ 여러 요소 하고싶음
 - 중복 가능, 한 요소가 여러 클래스 가능
- id 한 요소만 스타일링 하고싶음
 - 한 요소는 아디 하나만 가짐, 중복 불가
- 클래스 내 클래스 css 꾸미려면
 - `.movie .title { }`
- css 파일과 잇기
 - <link href="css/style.css" rel="stylesheet">
- 코멘트
 - <!-- 내용 --> html
 - /* */ css
- 개발자 도구, w3school, JSFiddle!
- `font-weight: 100단위 or bold,normal` (- 글씨 굵기)
- `text-align`
- `text-decoration`
- `px pt (절대적) / % (상대적 - 부모 요소 대비), em(1.5em = 150%)`
- `line-height = 줄 간격`
- `font-family : "이 글꼴 해보고", "안되면 이 글꼴"`
- `div span`
 - 묶어준다는 공통점

- div는 줄을 새로 바꿔버림, span은 그대로 놔둠

▼ 여행 사이트 코드

```
<!DOCTYPE html>
<html>
<head>
<title>travel</title>
<link href="styles.css" rel="stylesheet">
</head>
<body>

<div class="menu">
<a style="font-weight:bold" class="menuele" href="index.html">Home</a>
    위처럼 안하고 <>로 감싸도 됨, 자신 페이지는 href="#" 으로 하던데.,
    <a class="menuele" href="seoul.html">Seoul</a>
    <a class="menuele" href="tokyo.html">Tokyo</a>
    <a class="menuele" href="paris.html">Paris</a>
</div>



</body>
</html>
```

```
.logo {
    display: block;
    margin-left: auto;
    margin-right: auto;
    width: 165px;
    height: 58px;
    margin-top: 80px;
}

.menu {
    text-align: center;
    font-family: Helvetica Serif;           ->글꼴 "" 안붙여도될듯
    font-size: 16px;
    color: rgb(#88,89,91);                //##58595b 가능
    margin-top: 60px;
    margin-bottom: 60px;
}

.menu a {                                -> 따로 클래스 만들어 줄 필요 없이 이렇게 하면 됨
    margin-left :20px;
    text-decoration: none;
}

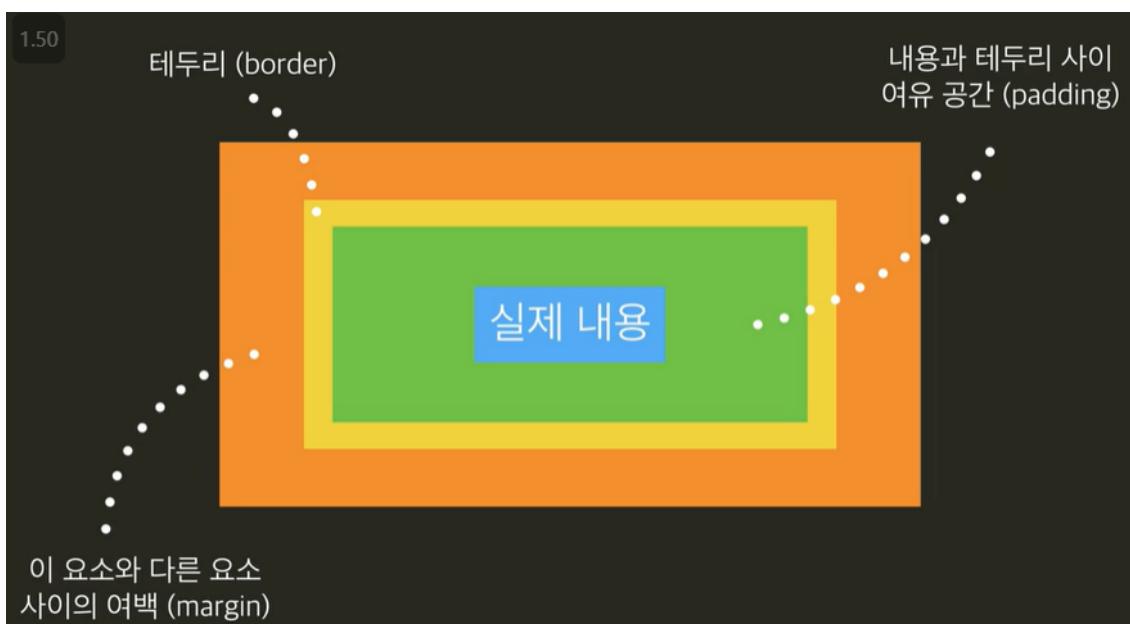
.imgs {
    display: block;
    margin-left: auto;
    margin-right: auto;
    width: 90%;
}
```

Travel

Home **Seoul** Tokyo Paris



- 박스 모델



- user agent stylesheet - 이것이 디폴트 css 값을 가짐 (개발자 도구)
- border : 5px solid red

- border-top / dotted, dashed, none
- padding : 50px 65px ~ (top부터 시계방향)
- margin-left right : auto = 둘다 오토로 하면 알아서 반반, 가운데 정렬됨
- min-width = 적어도 ~ 이어야 (창 사이즈 줄여도)
- overflow:
 - hidden - 보더 이후로는 안보임
 - visible - 디폴트, 그대로 넘쳐 보임
 - scroll, auto - 스크롤

▼ 자바스크립트 종급 (인터랙티브)

- html - js 연결
 - <script src="index.js"> html에 써주기
- document.getElementById('아이디값 문자열')
 - 없는 값 탐색 시 null
- getElementsByClassName()
 - 그 클래스 가진 태그들 모두 배열의 형태로 나옴 - 위부터 차례대로
 - 유사배열 (**HTMLCollection**) = 배열 형태지만 배열 메소드는 못씀(length, [] 접근, for of는 가능)
 - 없는 클래스 → [] 빈 배열 출력됨
- getElementsByTagName('button');
 - <button> 태그 이름으로 검색
 - (*)이면 모든 태그, 유사 배열로 리턴
- 하나의 이름으로 id class 모두 가능한 **css 선택자**로 (이게 더 유용)
 - querySelector('#mynumber') = 엘리먼트id와 동일
 - querySelectorAll('.클래스이름') = 클래스, []로 나옴?
 - all 안붙이면 클래스 중 첫번째 요소만,

이벤트

- 이벤트 핸들러 = function


```
b.onclick = function()
{ alert('정답입니다!👏') }
```
- window 객체 - js 최상단, 다른 모든 내장객체 포함 = 전역객체
- DOM - document obj model
 - html 안 문서 전체를 객체로 표현 / html을 js 관점에서 보는 느낌
 - console.dir()
 - log는 html 태그 형태, 파라미터로 전달받은 값 자체
 - dir은 객체 속성 자세히 표현, 달리 여러 개 표현 못함
 - 이를 이용해 js로 html 스타일을 수정, 설정 가능
- dom 트리
 - 요소 노드, 텍스트 노드
 - 자식, 부모, 형제 노드

프로퍼티	유형	결과
element.children	자식 요소 노드	element의 자식 요소 모음(HTMLCollection)
element.firstChild	자식 요소 노드	element의 첫 번째 자식 요소 하나
element.lastElementChild	자식 요소 노드	element의 마지막 자식 요소 하나
element.parentElement	부모 요소 노드	element의 부모 요소 하나
element.previousElementSibling	형제 요소 노드	element의 이전(previous) 혹은 좌측(left)에 있는 요소 하나
element.nextElementSibling	형제 요소 노드	element의 다음(next) 혹은 우측(right)에 있는 요소 하나

요소 노드 프로퍼티

- 태그.innerHTML = 'html코드'
 - 이런식으로 js에서 html 수정하기도
 - += 하면 추가 가능
 - innerText은 text만 취급
- outerHTML
 - inner + 포함하는 아우터 태그까지 포함해서
- textContent = “~”
 - inner와 비슷하지만, html <>등 제외한 텍스트만 가져옴
 - 할당 시 특수문자도 그냥 텍스트로 처리
 - innerText와 거의 동일
- 새로운 값을 할당할 경우 요소 자체가 교체되어 버리기 때문에 주의해야
- 요소 노드 추가 - 3스텝

```
// 1. 요소 노드 만들기: document.createElement('태그이름')
const first = document.createElement('li');

// 2. 요소 노드 꾸미기: textContent, innerHTML, ...
first.textContent = '처음';

// 3. 요소 노드 추가하기: NODE.prepend, append, after, before
tomorrow.prepend(first);
```

태그 만들고, 요소 콘텐츠 넣고, 특정 위치에 집어넣기

- 노드 삭제
 - tomorrow.remove() = 투모로우 ol 전부삭제
 - tomorrow.children[2].remove() = ol 3번째 삭제
- 노드 이동
 - 옮길곳.append(기존.children[2])
 - after, before는 기존의 sibling 위치에 이동시킴
- html 속성 다루기
 - href 등 비표준 속성 접근 불가 → 모두 접근 가능하도록
 - tomorrow.getAttribute('href')
 - tomorrow.href 안되고

- `setAttribute('속성', '값')` = 속성 추가(수정)
 - `removeAttribute('속성')` = 속성 제거
1. 속성에 접근하기: `element.getAttribute('속성')`
 2. 속성 추가(수정)하기: `element.setAttribute('속성', '값')`
 3. 속성 제거하기: `element.removeAttribute('속성')`

- 비표준 속성
 - `const fields = document.querySelectorAll('[field]');` 등등.. 있지만
 - 좀 더 안전하게, **dataset** 프로퍼티

`data-`로 시작하는 속성은 모두 `dataset`이라는 프로퍼티에 저장되는데요. 예를 들어서 `data-status`라는 속성이 있다면, `element.dataset.status`라는 프로퍼티에 접근해서 그 값을 가져올 수 있는 것 (일부러 `status`를 `data-status`로 고쳐서 사용하자)
- html 스타일 다루기
 - `.style.backgroundColor = '#000000'`

속성값은 -로 있는게 아니라 camel case 대문자 방식으로,
할당값은 - 있어도됨 그대로!
 - 이보단 그 태그의 **클래스 변경**이 권장됨
 - 태그.`className` = 'done'
 - 하지만 이러면 추가 이렇게 안되고 던으로만 바뀜
 - `.classList.add('done', 'other')`
 - `remove`
 - `.classList.toggle` - (클래스, true면 add false면 remove) - 클래스 하나만 다룬다는 특징
`//toggle은 한 속성을 on / off 하도록 하는 것`
 - ▼ 스타일 입히는 코드 모두 가능

```

item.style.opacity = '0.5';
item.style.textDecoration = 'line-through';

item.className = 'done';

item.classList.add('done');

item.setAttribute('class', 'done');

```

-
- 이벤트핸들러
 - `btn.onClick ~` 도 있지만 아래를 더 많이 씀
 - **addEventListener('click', event1)**

`event1`은 이미 정의해준 함수명, () 붙이면 안됨
같은거에 2개 핸들러 등록 시 최근꺼만 적용됨
 - `remove~동일`

`add`할때와 핸들러와 타입('click')이 동일해야만 삭제 가능
 - 마우스, 키보드, 포커스, 입력, 스크롤 이벤트 - 북마크
 - 두번째 파라미터에서 함수 표현이 아니라 `function() {}` 처럼 정의 하는 식이면 같은 형태라도 다른 함수
 - 이벤트 객체

■ 이벤트함수(event)

event.target 이벤트 발생한 요소로 많이 씀

```
const todoList = document.querySelector('#to-do-list');
const items = todoList.children;

function updateToDo(event) {
    event.target.classList.toggle('done')
}

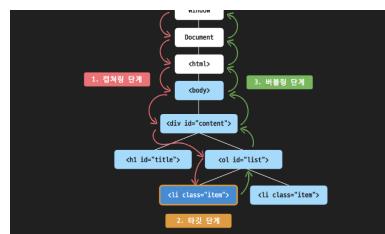
// 해당 요소의 속성에 접근해서, 클래스리스트에 done 속성을 키겠다!
// toggle은 한 속성을 on / off 하도록 하는 것
// removeEventListener 활용도하려면 이렇게 함수화 하는게 좋겠지

for(let i=0;i<items.length;i++){ //items.length - item은 유사배열
    items[i].addEventListener('click', updateToDo)
}

// 테스트 코드
items[2].removeEventListener('click', updateToDo);
```

○ 이벤트 버블링

- 같은 태입에 한하여 부모 요소까지 동작해버림
- event.target은 그 누른 요소 잘 출력,
currentTarget은 버블링 되는 그 요소 출력됨 (이벤트 핸들러가 등록된 요소)
- 막는 방법
event.stopPropagation()
- 웬만하면 잘 막을 일 없다. 막으면 나중에 전체 태그 body 등 실행 시 막은 부분 실행 안될듯 한 단점



이벤트가 발생하면 가장 먼저 `window` 객체에서부터 `target` 까지 이벤트 전파가 일어납니다. (캡처링 단계)
그리고 나서 타깃에 도달하면 타깃에 등록된 이벤트 핸들러가 동작하고, (타깃 단계)
이후 다시 `window` 객체로 이벤트가 전파됩니다. (버블링 단계)

○ 이벤트 위임

- 자식요소에 이벤트 개별적으로 다 달아준다면,
비효율적이고 나중에 추가된 요소는 반영 못함
- 버블링을 활용한 방법임 - 자식요소의 이벤트를 부모 요소에 위임 = 부모에 이벤트 할당 = 자식 모두 영향 받게됨
- but! 원하는 요소에서 일어나게끔 처리해줘야함
 - `tagname=li, contains('자식에만 있는 클래스')` 등의 방법으로

```
const todoList = document.querySelector('#to-do-list');

// 1. updateToDo 함수를 완성해 주세요
// item 클래스를 가진 리스트를 클릭시에만 동작하도록
function updateToDo(event) {
    if(event.target.classList.contains('item')) {
        event.target.classList.toggle('done')
    }
}

// 2. 각 li 태그가 아니라 하나의 태그에만 이벤트 핸들러를 등록해 주세요
todoList.addEventListener('click', updateToDo)

// 테스트 코드 - 새로 추가한 것도 반영되겠지, 부모 위임했으니
const newToDo = document.createElement('li');
newToDo.textContent = '가계부 정리하기';
```

```

newToDo.classList.add('item');
todoList.append(newToDo);

//버블링 막아놨으니 앤 done 추가 동작 안함
todoList.children[2].addEventListener('click', function(e) {e.stopPropagation()});

```

- 브라우저 기본 동작 - 디폴트

인풋 태그에 글씨 써지는, 마우스 오른 클릭 시 옵션 뜨는.

- event.preventDefault()

- 마우스 버튼 이벤트

- MouseEvent.button

0 1 2 - 왼 휠 오른

- .type

더블클릭시 순서 (순서도 중요)

mousedown mouseup click (x2)

dblclick

오른(contextmenu)

```

const flagBlue = document.querySelector('.flag-blue');
const flagWhite = document.querySelector('.flag-white');

function reset() {
    document.querySelector('.up').classList.remove('up');
}

function flagUp(e) { //마우스다운 해당하는 e 이벤트 받음
    if(e.button==0){ //target처럼 마우스다운 이벤트가 가지는 프로퍼티 button
        flagBlue.classList.toggle('up')
        //flagBlue.classList.add('up') 가능
        //이것의 의미는 flagblue 클래스 가진 애들에게 up이라는 클래스 하나 더 주는 것임
    }
    else if(e.button==2) {
        flagwhite.classList.toggle('up')
    }

    // 500 밀리초 뒤에 reset함수를 실행
    setTimeout(reset, 500);
}

document.addEventListener('contextmenu', function (event) {
    event.preventDefault() // 
});

// 테스트 코드 - 이벤트리스너에서 마우스다운 타입 설정
document.addEventListener('mousedown', flagUp);

```

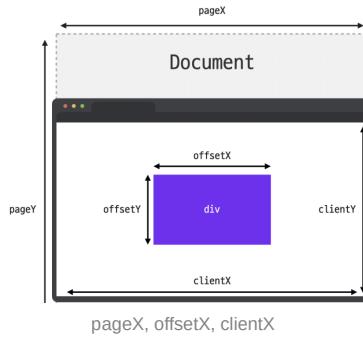
- 마우스 이동 이벤트

- e.type

- mousemove

box1.addEv('mousemove', onMouseMove)

위 온마우스무브 함수 내 e.clientX ~



- mouseover = 요소 밖에서 안으로 이동

mouseenter (동일 but 이벤트가 자식 요소에 영향 끼치는지 차이) - 자식 요소에 들어갈 때 한번 더 발생함, over는 함

- mouseout = 안 → 밖

mouseleave

- e.target = 이벤트가 발생한 요소

- e.relatedTarget = 이벤트 발생 직전or직후 마우스가 위치했던 요소

```
// showTitle 함수를 완성해 주세요
function showTitle(e) {
    if(e.target.dataset.title){ //마우스 올려놓은 곳, 버블링되어 자식까지 영향
        const span = document.createElement('span') //e.target.createElement('span') 아님
        span.classList.add('title')
        span.textContent=e.target.dataset.title //innerText
        e.target.append(span)
        //span은 표시되는 플로팅 텍스트를 띠워주기 위한 것임
    }
}

// removeTitle 함수를 완성해 주세요
function removeTitle(e) {
    if(e.target.dataset.title){
        e.target.lastElementChild.remove() //remove(lastElementChild) 아님
    }
}

// '대상'과 '타입'을 수정해 주세요
const map = document.querySelector('.map')
map.addEventListener('mouseover', showTitle);
map.addEventListener('mouseout', removeTitle);
// document도 가능
// 이벤트 위임이나 혹은 자식 요소의 영역에서도 이벤트가 발생하길 원할 때 mouseover/mouseout 타입의 이벤트를 활용
```

- 키보드 이벤트

- event.type

- keydown - 이걸 추천
- keypress - 출력값이 변하는 때에만 발생, 꾹 누르는 동안엔 한번만 발생
- keyup

- event.key

어떤 값을 눌렀는지

- event.code

어떤 위치를 눌렀는지, key와 동일할 수도 다를 수도

```
const chatBox = document.querySelector('#chat-box');
const input = document.querySelector('#input');
const send = document.querySelector('#send');

function sendMyText() {
```

```

const newMessage = input.value;
if (newMessage) {
  const div = document.createElement('div');
  div.classList.add('bubble', 'my-bubble');
  div.innerText = newMessage;
  chatBox.append(div);
} else {
  alert('메시지를 입력하세요... ');
}
input.value = '';
}

send.addEventListener('click', sendMyText);

// 여기에 코드를 작성하세요
function sendMyText2(e){
  if(e.key === 'Enter' && !e.shiftKey){ //엔터 누르되 쉬프트는 안눌렸을때 전송 - 쉬프트를 누르면 본래 엔터인 줄바꿈이 되겠지 - 쉬프트 누르면 이곳에
    //이벤트가 발생할 때 shift키를 눌렀는지를 불린 형태로 담고 있어요.
    sendMyText();
    e.preventDefault();
    //원래 textarea 태그에 커서를 두고 enter키 버튼을 누르면 줄바꿈이 됩니다. 이건 브라우저의 기본 동작
  }
}
input.addEventListener('keypress', sendMyText2)
// 인풋창에 엔터를 누르는걸 기다리고 있으므로

```

- **input 태그 타입**

#form.addEve('focusin', ~)

- 포커스 이벤트

- focusin - 요소에 포커스

- focus - 버블링x

- focusout - 요소에 포커스 빠져나갈 때

- blur - 버블링x

- 입력 이벤트

- input - 태그 안에 입력 - esc등은 반응x

- change - 포커스아웃 등으로 입력 끝났다 판단 이후 값 변경 시

- select - 입력 양식의 하나 선택

- submit - 폼 전송

```

const input = document.querySelector('#input');

function checker() {
  const words = document.querySelectorAll('.word');
  if (words.length === 0) {
    alert('Success! 🎉');
    if(confirm('retry?')) {
      window.location.reload();
    }
  }
}

// 여기에 코드를 작성하세요
function func1(e) {
  const words = document.querySelectorAll('.word'); //word class 가진 태그 다 가져오고
  for(let i=0;i<words.length;i++){ //그 태그들 다 돌면서
    if(input.value === words[i].dataset.word){ //인풋 값이 그 태그들에 있으면
      input.value = '';
      words[i].remove();
      checker()
    }
  }
}
//input 값 변화가 생길때마다 함수로 체크
input.addEventListener('change', func1)

```

- **스크롤 이벤트**

```

// Scroll 이벤트
let lastScrollY = 0;

function printEvent(e) {
  const STANDARD = 30;

  const nav = document.querySelector('#nav');
  const toTop = document.querySelector('#to-top');

  if (window.scrollY > STANDARD) { // 스크롤이 30px을 넘었을 때
    nav.classList.add('shadow');
    toTop.classList.add('show');
  } else { // 스크롤이 30px을 넘지 않을 때
    nav.classList.remove('shadow');
    toTop.classList.remove('show');
  }

  if (window.scrollY > lastScrollY) { // 스크롤 방향이 아래쪽 일 때
    nav.classList.add('lift-up');
  } else { // 스크롤 방향이 위쪽 일 때
    nav.classList.remove('lift-up');
  }
}

lastScrollY = window.scrollY;

window.addEventListener('scroll', printEvent);

```

window.add 많이씀, scrollY 프로퍼티로 스크롤된 특정 위치 기준 활용

▼ 모던 자바스크립트

= es3,4,5,6 .. 등 현시점 사용하기 적합한 범위 내 최신 버전 표준을 준수하는 js

es6(es2015)는 에크마스크립트 버전 중 js 발전에 가장 큰 영향을 준 버전

JavaScript는 **프로그래밍 언어**이고, ECMAScript는 **프로그래밍 언어의 표준**

- 데이터 타입 - 유연한 js의 데이터 타입 특징! - typeof로 확인!
 - 넘버
 - 스트링
 - 불린
 - null
 - undefined
 - symbol

▼ 코드 내에서 유일한 값을 가진 변수 이름을 만들 때 사용

심지어는 똑같은 설명을 붙인 심볼을 만들더라도 두 값을 비교하면 false가 반환됩니다.

```

const symbolA = Symbol('this is Symbol');
const symbolB = Symbol('this is Symbol');

console.log(symbolA === symbolB); // false

```

- BigInt

- object (위에는 기본형, 이거는 참조형)
- 다른 타입 → 불린 형변환

for if 등 불린 값이 들어가야 하는 자리에

false null und nan 0 “ = false

{ }, [] 포함 모두 = true
- AND OR 연산 방식
 - 직관적으로 t f 생각하면 되는데 둘다 일 경우 어떤게 리턴될지 보려면 알아야함, 기본적 and가 우선순위

true && false =

 - 왼쪽 값이 true면 오른쪽 리턴
 - 왼 f 면 왼 리턴

true || false

 - 왼 t면 왼 리턴
 - 왼 f면 오른 리턴

```
// AND와 OR의 연산 방식
console.log(null && undefined); // null
console.log(0 || true); // true
console.log('0' && NaN); // NaN
console.log({} || 123); // {}
```

- ?? 연산자

null 혹은 undefined 값을 가려내는 연산자

```
const title1 = false || 'codeit'; //codeit
const title2 = false ?? 'codeit'; //false
```
- var 문제
 - 호이스팅 되어버림 (선언문만)

선언 전에 사용했을 때 에러가 안뜸
 - 변수 이름 중복 선언이 되어버림, 이전 값이 사라짐
 - 스코프의 차이 - 조건문 반복문 안에서 선언되어도 전역변수가 되어버림, 지역변수가 안됨, 함수에서는 됨
함수스코프 (VS 블록 스코프)

- 함수표현식

addevent에서처럼 두번째 파라미터에서 fun() { 이것도 해당

 - 함수선언을 값처럼 사용한다는 점
 - var에 할당 시 함수 스코프, let const에 할당하면 블록 스코프
 - function a() { } 이게 함수선언식
이건 호이스팅 되어서 어디든 동작, 함수 스코프 (함수 내에서 선언 시 외부에서 사용 불가)
- 이름 있는 함수표현식
- 선언과 동시에 즉시 실행 함수
- 값으로서 함수
 - 객체 프로퍼티 중 하나로 선언

- 배열 중 하나로 선언
- 다른 함수의 파라미터에 전달된 함수 = 콜백함수
- 함수의 리턴을 함수 = 고차함수

- 파라미터 = 함수 선언 시, 아규먼트 = 함수 호출 시 전달하는 값
- 파라미터에 기본값 주기 (name = 'codeit') = 이런 형식이라면 마지막 파라미터에 주길
 - undef로 기본값 그대로 사용 가능
 - (x, y=x+3) 등으로 활용 가능

- 아규먼트 개수 따라 전달 받은 개수 따라 유연하게 출력하게 하는 법
 - arguments 객체 = 유사 배열
 - for(let arg of arguments) { log(arg) }

```
function fun1() { ~~~~~ //파라미터 없음, rest는 있음
let word = '';

for(const arg of arguments) { //모든 아규먼트의 하나씩 접근
  word = word.concat(arg[0]) //각 아규먼트의 첫 글자
  // word += arg[0] 와 동일한 기능
}
```

- Rest parameter = arguments 객체보다 권장
 - 다른 파라미터와 같이 사용 가능

```
// Rest Parameter
function printRank(first, second, ...others) {
  console.log('코드잇 레이스 최종 결과');
  console.log(`우승: ${first}`);
  console.log(`준우승: ${second}`);
  for (const arg of others) {
    console.log(`참가자: ${arg}`);
  }
}

printRank('Phil', 'Won', 'Emma', 'Min', 'Luke');

function ignoreFirst (...args) {
  for(const arg of args){
    if(arg == args[0]) continue;
  }
}
```

- arrow func

보다 간결하게, 익명 함수 ⇒ 변수에 할당 or 다른 함수를 호출할 때 아규먼트로 사용됨

() => {} //function 지우고 => 추가

파라미터 하나면 소괄호 생략하기도, 리턴문에서 return도 생략하기도...

```
const getObject = (a, b, c) => ({ 0: a, 1: b, 3: c });

arguments 객체가 존재 x

this가 선언 직전 객체로 인식 - 객체 메소드 만들 땐 일반 함수표현식 쓰자
```

- this

함수를 호출한 객체를 가리킴

함수 호출 시 결정되는 값, 객체 내 함수가 정의 되어있으면 그 객체 출력

- 문장과 표현식
- 조건 연산자 - if else 문을 간단하게 표현
 - 조건 ? (트루면 하는) : (펄스면 하는)
 - (표현식)이라 변수 선언, 반복문 선언 등은 못함

```

let msg = '';
if (x > 3) { msg = 'x는 3보다 크다.'; }
else { msg = 'x는 3보다 크지 않다.'; }
----- 변환 시
let msg = (x > 3) ? 'x는 3보다 크다.' : 'x는 3보다 크지 않다.';

이렇게도 쓰인다
const passChecker = (score) => score > cutoff ? '합격입니다!' : '불합격입니다!';
console.log(passChecker(75));

```

- spread = 둑여있는 배열을 하나의 원소로 펼치는 것

numbers = [1,2,3]

...numbers ⇒ 1,2,3 (= 여러 값의 목록으로 간주)

```

const webPublishing = ['HTML', 'CSS'];
const interactiveWeb = [...webPublishing, 'JavaScript'];

```

- 배열 복사

배열은 주소 복사라

a = b.slice() 해야하는데

a = [...b] 가능,

a = [...b, '추가할 요소'] 도 바로 가능

- 배열 두 개 붙이기

c = a.concat(b) 과 동일하게

c = [...a, ...b] 가능

- 파라미터 값 전달

func(...myArr) myArr = [2017, 'codeit']

- 배열을 펼쳐 객체에 담기

a = { ...myArr }

key가 index 0,1,2 되고, value는 어레이 값이 됨

```

const members = ['태호', '종훈', '우재'];
const newObject = { ...members };
console.log(newObject); // {0: "태호", 1: "종훈", 2: "우재"}

```

- 객체를 복사하거나 기존의 객체를 가지고 새로운 객체를 만들 때

```

const latte = {
  espresso: '30ml',
  milk: '150ml'
};

const cafeMocha = {
  ...latte,
  chocolate: '20ml',
}

```

```

const snacks = ['원카칩', '꿀버터칩', '헛스윙칩', '태양칩', '야채시간'];
const drinks = ['사이다', '콜라', '우유', '물', '커피', '레몬에이드'];

function printArguments(...args) {

```

```

        for (const arg of args) {
            console.log(arg);
        }
    }

    // 1. Spread 구문을 활용해서 snacks와 drinks 배열을 각각 mySnacks와 myDrinks 변수에 복사해 주세요
    const mySnacks = [...snacks]
    const myDrinks = [...drinks]

    mySnacks.splice(2, 3);
    myDrinks.splice(1);

    // 2. Spread 구문을 활용해서 mySnacks와 myDrinks 순서로 두 배열을 합쳐서 myChoice 변수에 할당해 주세요
    const myChoice = [...mySnacks, ...myDrinks]

    // 3. Spread 구문을 활용해서 myChoice의 각 요소들을 printArguments 함수의 아규먼트로 전달해 주세요
    printArguments(...myChoice);

```

- 모던! 객체 프로퍼티

- 키와 밸류 이름이 동일하다면 그냥 title 하나만 표시
- 객체 내 getFullname : function() {}
getFullscreen() {}로 축약하기도 함
- 계산된 속성명 (key)
['aa' + 'b'] [time] [함수() 콜 해서 리턴값] -모두 가능

- 읍셔널 체이닝

- 접근하려는 객체가 예상한 프로퍼티를 가지고 있지 않을 수

```

function printCatName(user) { console.log(user.cat?.name); }

----- 위와 동일한 의미
console.log((user.cat === null || user.cat === undefined) ? undefined : user.cat.name);

```

- 구조 분해 (Destructure)

- 배열
각 원소에 값 할당하려면 const mac = rank[0] 이렇게 다 해줘야하는데
const rank = [1,2,3,4]
const [mac, ipad ..] = rank 하면 한번에 됨
 - 위에서 마지막에 [, ...coupon] = rank 하면 나머지는 모두 쿠폰으로
coupon[2] 이런 식으로 접근 가능
 - rank가 짧으면 undef 할당, 기본값 넣을 수 있음
- 교환
원래는 temp로 3줄 써야하지만
[mac, ipad] = [ipad, mac] 가능 - 들에 담긴 값이 교환됨

```

// 1. Destructuring 문법을 활용해서 numbers 배열의 각 요소를 one, two, three라는 변수에 할당해보세요
const numbers = [1, 2, 3];
const [one, two, three] = numbers

// 2. Destructuring 문법을 활용해서 TV는 livingRoom, 나머지 요소들(배열)은 kitchen 변수에 할당해 주세요
const products = ['TV', '식탁', '냉장고', '전기밥솥', '전자레인저', '오븐', '식기세척기'];
const [livingRoom, ...kitchen] = products

// 3. Destructuring 문법을 활용해서 두 변수의 값을 서로 바꿔주세요
let firstName = 'Kang';

```

```
let lastName = 'Young';
[firstName, lastName] = [lastName, firstName]
```

- 객체

기존 const title = macbook.title

const {title, color} = macbook 으로 한번에 선언 가능

- 없는 프로퍼티 undef, 기본값 가능

- {title: title1, ...rest}로 선언 가능

- 객체 내부 프로퍼티 이름이 아닌 새로운 이름인 title1으로 사용 가능

- 이것도 ...rest로 title 제외 나머지 프로퍼티들을 한 묶음 객체로 사용 가능

```
// 1. Destructuring 문법을 사용해서 title, artist, year, medium 변수에 myBestArt 객체의 각 프로퍼티를 할당해 주세요
const myBestArt = {
  title: '별이 빛나는 밤에',
  artist: '빈센트 반 고흐',
  year: 1889,
  medium: '유화',
};
const {title, artist, year, medium} = myBestArt

// 2. Destructuring 문법을 활용해서 myBestSong의 프로퍼티 중 title과 artist는 각각 songName과 singer라는 변수에, 나머지는 rest라는 변수에
const myBestSong = {
  title: '무릉',
  artist: '아이유(IU)',
  release: '2015.10.23.',
  lyrics: '모두 점드는 밤에...'
};
const {title : songName, artist : singer, ...rest} = myBestSong
//기존 프로퍼티 네임과 다르게 사용, 나머지는 rest라는 변수에 객체로 할당

// 3. printMenu 함수 안에 잘못 작성된 Destructuring 코드를 수정해 주세요
const menu1 = { name: '아메리카노' };
const menu2 = { name: '바닐라 라떼', ice: true };
const menu3 = { name: '카페 모카', ice: false };

function printMenu(menu) {
  // menu 파라미터로 전달되는 객체에 ice 프로퍼티가 없을 경우 기본값은 true
  const {name, ice=true} = menu
  console.log(`주문하신 메뉴는 ${ice ? '아이스' : '따뜻한'} ${name}입니다.`);
}
```

- 함수

- 대표적으로,

```
btn.addEventListener('click', (event) => { event.target.classList ~ } )
```

에서 $\{ \{ \text{target} \} \} \Rightarrow \{ \text{target}.classList ~ \}$

- 정확한 모르겠지만 함수 파라미터 내에서 객체 분해?

```
function printFavoritSong(name, music) {
  console.log(`최근 '${name}'님이 즐겨듣는 노래는 '${music.singer}'의 '${music.title}'이라는 노래입니다.`);
}

function printFavoritSong(name, {title, singer}) {
  console.log(`최근 ${name}님이 즐겨듣는 노래는 ${title}의 ${singer}이라는 노래입니다.`)
}

function printFavoritSong(name, { title, singer }) {
  console.log(`최근 '${name}'님이 즐겨듣는 노래는 '${singer}'의 '${title}'이라는 노래입니다.`);
}
```

- null은 값이고 undefined는 말그대로 할당될 값이 없는 걸 뜻함

- 에러

- 선택스에러(문법에 맞지않은 에러 - 아예 아무것도 실행 x), 타입에러(잘못된 방법 자료형), 레퍼런스에러(존재x인걸 사용하는 에러)

- 에러 객체 = {name, message}

- 우리가 에러 객체 만들기

```
const error = new TypeError('발생!')
error.name = TypeError
error.message = 발생!

throw error; ⇒ 에러 발생 시킴, 의도적으로!
    ■ throw new Error('발생~')


```

- try catch

```
// try catch 문
try {
    console.log('에러 전');

    const codeit = '코드잇';
    console.log(codeit)

    codeit = 'codeit';

    const language = 'JavaScript';
    console.log(language);
} catch (e) {
    console.log('에러 후');
    console.error(e);
    console.log(e.name);
    console.log(e.message);
}
```

lang 부분은 동작x

- try문 내부 중간에서 에러 발생 시 try 이후는 실행 안되지만, for문에서 돌리면 1에서 에러나도 2도 실행되는 장점, 안정적이다
- try catch도 블록이라 const let 선언시 서로 사용 안됨 주의
- try 문에서 에러가 발생해서 에러 객체가 만들어지면, 그 에러 객체를 catch 문 안에서 다룰 수
- Finally

try문에서 어떤 코드를 실행할 때 에러 여부와 상관 없이 항상 실행할 코드를 작성

```
try {
    // 실행할 코드
} catch (err) {
    // 에러가 발생했을 때 실행할 코드
} finally {
    // 항상 실행할 코드
}
```

• 배열만의 메소드

- 배열의 단순 반복작업 = forEach

배열 반복작업으로 새로운 배열을 뽑기 위한 = map

- forEach

- for of문 사용했지만 이것도 있다

```
const members = [1,2,3,4]

for(let member of members) {
    console.log(member)
}

//for of 문을 forEach로 전환함
members.forEach(function (member){
    console.log(member)
}))
```

```
//애로우 평선으로 전환함
members.forEach((member, index, arr) => {
    console.log(index, member, arr)
})

//member는 배열의 요소가 전달됨
//index는 각 요소의 인덱스
//arr는 반복중인 배열 자체 = members, 잘 사용은 안됨
```

- 기본적으로 배열의 개수만큼 반복함
- members.pop() 처럼 반복 중에 배열의 길이가 줄어들면 반복도 그만큼 줄어듬
- members.push()로 배열 길이 늘어나는건 기존 member 배열 길이만큼만 반복함
- 예제 - 숨겨진 단어 찾기 - 행으로 쌓을때 young 찾기

```
const quiz = ['YUMMY', 'COUNT', 'ABUSE', 'SOUND', 'SWING'];

// 여기에 코드를 작성하세요
const answer = quiz.map((char, i) => {
    return char[i]
})
//더 간결한 답
const answer = quiz.map((char, i) => char[i])
```

◦ map

- 설명

```
//map은 함수 결과로 새로운 배열이 리턴됨
//그거랑 forEach와 동일, 위에 map 넣어도 그대로 동작
const firstName = [1, 2, 3, 4]
const lastName = [10, 20, 30, 40]

firstName.map((names, i) => {
    console.log(names, lastName[i])
})

const fullName = firstName.map((names, i) => {
    return names + lastName[i]
})

//위를 애로우로 간결하게
const fullName = firstName.map((names, i) => names + lastName[i])
console.log(fullName)
```

- 예제 - 투두리스트 - data에 있는 일 html에 투두로 등록하기

```
const list = document.querySelector('.list');
const data = [
    {
        title: '자바스크립트 공부하기',
        isClear: true,
    },
    {
        title: '쓰레기 분리수거',
        isClear: false,
    }
];

// 여기에 코드를 작성해 주세요.
data.forEach((todo, index) => { //todo는 요소 하나하나 (title,isClear 포함한 객체 하나)
    const li = document.createElement('li'); //li = document.createElement 주의, li 태그 만들
    li.textContent = `${index+1}. ${todo.title}`
    li.classList.add('item')
    if(todo.isClear == true) {
        li.classList.add('done')
    }
    list.append(li) //append로 li를 어디에 넣을지 정함
})
```

◦ filter

- map 메소드와 형태는 비슷, map은 리턴문에 각 요소에서 필요한 값을 써주는데,
- filter는 불린 조건식을 써주면 그에 해당하는 요소들이 배열 형태로 출력됨

```

const seoul = ['김영훈', '김윤수', '김동욱', '강대위']

const notKims = seoul.filter((names)=>{
    return names[0]!='김'
})
----- 이렇게 쓰기도 연습
const notKims = seoul.filter((names)=>names[0]!='김')

```

- o find

filter랑 거의 비슷

- filter는 배열, find는 값
- 반복횟수 차이 = find는 그 조건인거 그 하나 찾으면 반복 그만둠

```

const nameInput = document.querySelector('#user-name');
const phoneInput = document.querySelector('#phone-number');
const findBtn = document.querySelector('#find');

const data = [ //전역변수로 선언한 객체의 배열
    { userName: '막대기', phoneNumber: '01012341111', email: 'stick@go_do_it.kr' },
    { userName: 'young', phoneNumber: '01012342222', email: 'kang@go_do_it.kr' },
];

function findEmail() {
    const nameValue = nameInput.value; //html에서의 input 위치 가져와서 그 인풋의 값 가져옴
    const phoneValue = phoneInput.value;

    // 여기에 코드를 작성하세요
    const user = data.find((info)=>{ //배열에서 각 요소마다 접근
        return (info.userName==nameValue && info.phoneNumber==phoneValue) //이러한 조건인 요소 객체 통째로 반환 => user
    })

    const message = user
        ? `${user.userName}님의 이메일은 ${user.email} 입니다. `
        : '이메일을 찾을 수 없습니다. 입력 정보를 다시 확인해 주세요.';

    alert(message);
}

findBtn.addEventListener('click', findEmail); //find 클릭 누르면 위 함수 발동

```

- o some

배열에서 조건 만족하는 요소가 하나라도 있는지 t or f

하나라도 찾으면 반복 종료

- o every

배열에서 모든 요소가 조건에 다 만족하는지 t or f

모든 요소 다돌아야 하므로 반복 계속

- 빈 배열이면 some = f // every = t

```

numbers = [1,2,5,7]
const everyReturn = numbers.every((el)=>{
    return el > 5
})

console.log(everyReturn) //false

```

```

const spait = [
    { codeName: 'ApplePie', members: ['스파이', '스파이', '스파이', '스파이', '스파이'] },
    { codeName: 'BigBoss', members: ['스파이', '스파이', '스파이', '스파이', '스파이'] },
    { codeName: 'CEO', members: ['스파이', '스파이', '스파이', '습하이', '스파이'] },
]

function checkSpy(team) { //team은 스파잇 배열에서 각 요소 하나하나
    // 여기에 코드를 작성하세요
    const every = team.members.every((a)=>{
        return a=='스파이' //이 조건이 맞는지 every 요소 체크해봐라
    })
}

if(every==false) message=[주의!] 팀 ${team.codeName} 에 이중 스파이가 있습니다!

```

```

        else message = `팀 ${team.codeName} 에는 이중 스파이가 없습니다.`  

        console.log(message);  

    }  
  

    spait.forEach((team) => checkSpy(team));  

//위 spait 배열을 forEach로 각 요소를 파라미터에 넣어 checkSpy 함수 돌림

```

- reduce

- 이전까지와 좀 다른 형태
- acc 파라미터 = 현재 콜백함수 리턴값이 다음 실행될 함수의 파라미터로 넘어감 accumal~
- 결국 최종 리턴값은 마지막 요소가 넘겨줄 다음요소가 없어서 주는 acc 값
- 처음 값은 넘겨받는 게 없으니 두번째 파라미터로 초기값 (0) 줌, 안주면 디폴트 첫 el 값

```

const numbers = [1,2,3,4]  
  

const all = numbers.reduce((acc, el, i, arr)=>{  

    console.log(i)  

    console.log(el)  

    console.log(acc)  

    return el + acc  

},0) //  
  

console.log(all)

```

- 예제 - 경력개월수합

```

const data = [  

    { company: 'Naber', month: 3 },  

    { company: 'Amajohn', month: 12 },  

];  
  

// 여기에 코드를 작성하세요  

const totalCareer = data.reduce((acc,el,i)=>{  

    return acc+(el.month)  

},0)  
  

console.log(`상원이의 경력은 총 ${totalCareer}개월입니다.`);

```

- sort

arr.sort() - 유니코드 순서 정렬, 문자열은 abc 순서

- 숫자 [1, 10, 21, 36000, 4] 이런식으로 나옴
 - 오름차순 ⇒ arr.sort((a, b) => a - b)
- 원본 배열 요소 정렬하므로 안전히 다른 변수에 복사 후 하자

- reverse

numbers.reverse()

원본 배열을 뒤집음

- ▼ map, set

Map과 Set

객체는 property name을 통해 이름이 있는 여러 값을 묶을 때 활용할 수 있고,
배열은 index를 통해 순서가 있는 여러 값을 묶을 때 유용하게 활용할 수 있습니다.

그런데 ES2015에서 객체와 비슷한 Map과 배열과 비슷한 Set이라는 데이터 구조가 새롭게 등장했는데요.
각각 어떤 특징들을 가지고 있는지 간단하게 살펴보도록 합시다.

Map

Map은 이름이 있는 데이터를 저장한다는 점에서 객체와 비슷합니다.

하지만, 할당연산자를 통해 값을 추가하고 점 표기법이나 대괄호 표기법으로 접근하는 일반 객체와 다르게

Map은 메소드를 통해서 값을 추가하거나 접근할 수 있는데요.

`new` 키워드를 통해서 Map을 만들 수 있고 아래와 같은 메소드를 통해 Map 안의 여러 값을 다룰 수 있습니다.

- `map.set(key, value)`: key를 이용해 value를 추가하는 메소드.
- `map.get(key)`: key에 해당하는 값을 얻는 메소드. key가 존재하지 않으면 `undefined`를 반환.
- `map.has(key)`: key가 존재하면 `true`, 존재하지 않으면 `false`를 반환하는 메소드.
- `map.delete(key)`: key에 해당하는 값을 삭제하는 메소드.
- `map.clear()`: Map 안의 모든 요소를 제거하는 메소드.
- `map.size`: 요소의 개수를 반환하는 프로퍼티. (메소드가 아닌 점 주의! 배열의 `length` 프로퍼티와 같은 역할)

```
// Map 생성
const codeit = new Map();

// set 메소드
codeit.set('title', '문자열 key');
codeit.set(2017, '숫자형 key');
codeit.set(true, '布尔形 key');

// get 메소드
console.log(codeit.get(2017)); // 숫자형 key
console.log(codeit.get(true)); // 布尔形 key
console.log(codeit.get('title')); // 문자열 key

// has 메소드
console.log(codeit.has('title')); // true
console.log(codeit.has('name')); // false

// size 프로퍼티
console.log(codeit.size); // 3

// delete 메소드
codeit.delete(true);
console.log(codeit.get(true)); // undefined
console.log(codeit.size); // 2

// clear 메소드
codeit.clear();
console.log(codeit.get(2017)); // undefined
console.log(codeit.size); // 0
```

문자열과 심볼 값만 key(프로퍼티 네임)로 사용할 수 있는 일반 객체와는 다르게

Map 객체는 메소드를 통해 값을 다루기 때문에, 다양한 자료형을 key로 활용할 수 있다는 장점이 있습니다.

Set

Set은 여러 개의 값을 순서대로 저장한다는 점에서 배열과 비슷합니다.

하지만, 배열의 메소드는 활용할 수 없고 Map과 비슷하게 Set만의 메소드를 통해서 값을 다루는 특징이 있는데요.

Map과 마찬가지로 `new` 키워드로 Set을 만들 수 있고 아래와 같은 메소드를 통해 Set 안의 여러 값을 다룰 수 있습니다.

- `set.add(value)`: 값을 추가하는 메소드. (메소드를 호출한 자리에는 추가된 값을 가진 Set 자신을 반환.)
- `set.has(value)`: Set 안에 값이 존재하면 `true`, 아니면 `false`를 반환하는 메소드.
- `set.delete(value)`: 값을 제거하는 메소드. (메소드를 호출한 자리에는 셋 내에 값이 있어서 제거에 성공하면 `true`, 아니면 `false`를 반환.)
- `set.clear()`: Set 안의 모든 요소를 제거하는 메소드.
- `set.size`: 요소의 개수를 반환하는 프로퍼티. (메소드가 아닌 점 주의! 배열의 `length` 프로퍼티와 같은 역할)

```
// Set 생성
const members = new Set();

// add 메소드
members.add('영훈'); // Set(1) {"영훈"}
members.add('윤수'); // Set(2) {"영훈", "윤수"}
members.add('동욱'); // Set(3) {"영훈", "윤수", "동욱"}
members.add('태호'); // Set(4) {"영훈", "윤수", "동욱", "태호"}

// has 메소드
console.log(members.has('동욱')); // true
```

```

console.log(members.has('현승')); // false

// size 프로퍼티
console.log(members.size); // 4

// delete 메소드
members.delete('충훈'); // false
console.log(members.size); // 4
members.delete('태호'); // true
console.log(members.size); // 3

// clear 메소드
members.clear();
console.log(members.size); // 0

```

한 가지 특이한 점은 일반 객체는 프로퍼티 네임으로, Map은 `get` 메소드로, 그리고 배열은 `index`를 통해서 개별 값에 접근할 수 있었는데요.

한 가지 특이한 점은 Set에는 개별 값에 바로 접근하는 방법이 없다는 점입니다.

```

// Set 생성
const members = new Set();

// add 메소드
members.add('영훈'); // Set(1) {"영훈"}
members.add('윤수'); // Set(2) {"영훈", "윤수"}
members.add('동욱'); // Set(3) {"영훈", "윤수", "동욱"}
members.add('태호'); // Set(4) {"영훈", "윤수", "동욱", "태호"}

for (const member of members) {
  console.log(member); // 영훈, 윤수, 동욱, 태호가 순서대로 한 줄 씩 콘솔에 출력됨.
}

```

그래서 위 코드와 같이 반복문을 통해서 전체 요소를 한꺼번에 다룰 때 반복되는 그 순간에 개별적으로 접근할 수가 있습니다. 그런데, 이런 특징을 가지고도 Set이 유용하게 사용되는 경우가 있는데요.

바로, 중복을 허용하지 않는 값들을 모을 때입니다.

Set은 중복되는 값을 허용하지 않는 독특한 특징이 있는데요.

```

// Set 생성
const members = new Set();

// add 메소드
members.add('영훈'); // Set(1) {"영훈"}
members.add('윤수'); // Set(2) {"영훈", "윤수"}
members.add('영훈'); // Set(2) {"영훈", "윤수"}
members.add('영훈'); // Set(2) {"영훈", "윤수"}
members.add('동욱'); // Set(3) {"영훈", "윤수", "동욱"}
members.add('동욱'); // Set(3) {"영훈", "윤수", "동욱"}
members.add('태호'); // Set(4) {"영훈", "윤수", "동욱", "태호"}
members.add('동욱'); // Set(4) {"영훈", "윤수", "동욱", "태호"}
members.add('태호'); // Set(4) {"영훈", "윤수", "동욱", "태호"}
members.add('태호'); // Set(4) {"영훈", "윤수", "동욱", "태호"}

```

최초에 추가된 순서를 유지하면서, 나중에 중복된 값을 추가하려고 하면 그 값은 무시하는 특징이 있습니다.

처음 Set을 생성할 때 아규먼트로 배열을 전달할 수도 있는데요.

이런 특징을 활용해서 배열 내에서 중복을 제거한 값들의 묶음을 만들 때 Set을 활용하기도 합니다.

```

const numbers = [1, 3, 4, 3, 3, 2, 1, 1, 1, 5, 5, 3, 2, 1, 4];
const uniqNumbers = new Set(numbers);

console.log(uniqNumbers); // Set(5) {1, 3, 4, 2, 5}

```

- Map

- 객체와 비슷하지만, 문자열과 심볼 값만 key(프로퍼티 네임)로 사용할 수 있는 일반 객체와는 다르게 Map 객체는 메소드를 통해 값을 다루기 때문에, 다양한 자료형을 key로 활용할 수 있다는 장점

`const map = new Map()`

- `map.set(key, value)`: key를 이용해 value를 추가하는 메소드.

- `map.get(key)`: key에 해당하는 값을 얻는 메소드. key가 존재하지 않으면 `undefined`를 반환.
 - `map.has(key)`: key가 존재하면 `true`, 존재하지 않으면 `false`를 반환하는 메소드.
 - `map.delete(key)`: key에 해당하는 값을 삭제하는 메소드.
 - `map.clear()`: Map 안의 모든 요소를 제거하는 메소드.
 - `map.size`: 요소의 개수를 반환하는 프로퍼티. (메소드가 아닌 점 주의! 배열의 `length` 프로퍼티와 같은 역할)
- Set
 - 여러 개의 값을 순서대로 저장한다는 점에서 배열과 비슷
 - 개별 값에 바로 접근하는 방법이 없다
`for (const member of members)` 같이 전체요소 한꺼번에 다루는 반복되는 순간에 개별적 접근 가능함 ;;
 - 유용! - 중복되는 값을 허용하지 않는 독특한 특징
`const numbers = [1, 3, 4, 3, 3, 3, 2, 1, 1, 1, 1, 5, 5, 3, 2, 1, 4];`
`const uniqNumbers = new Set(numbers); // {1,2,3,4,5}`
- ```
const set= new Set();

 - set.add(value): 값을 추가하는 메소드. (메소드를 호출한 자리에는 추가된 값을 가진 Set 자신을 반환.)
 - set.has(value): Set 안에 값이 존재하면 true, 아니면 false를 반환하는 메소드.
 - set.delete(value): 값을 제거하는 메소드. (메소드를 호출한 자리에는 셋 내에 값이 있어서 제거에 성공하면 true, 아니면 false를 반환.)
 - set.clear(): Set 안의 모든 요소를 제거하는 메소드.
 - set.size: 요소의 개수를 반환하는 프로퍼티. (메소드가 아닌 점 주의! 배열의 length 프로퍼티와 같은 역할)
```

- 모듈화

코드가 길어지면 기능마다 파일을 나눔

재사용 가능, 효율적 관리 가능

- 표준화된 모듈 문법 (javascript에서)

- 모듈 스코프

하나의 html을 공유하는 js 2 파일에서 같은 변수를 공유하는 문제

⇒ `<script type="module" src="index.js">` 하면 공유 못하게

- export

다른 파일에서도 사용 가능토록, 변수나 함수 선언문 앞에 써라

▪ 쓸곳에서 위에 써줘야

```
import { 변수나 함수 등, ○○ } from '경로';
```

▪ 이런식으로 쓰면 html에 script 태그도 js 중 한 파일만 연결해놓으면 됨

```
-----index.js
import {addMenu} from './add.js'
// addMenu 함수 안에 add, emptyAlert 등의 함수가 쓰이지만 이거만 임포트 해도 됨
// 임포트해도 여기서 실행이 아닌 add.js에서 실행됨, 연결다리만 놓은 셈

const data = []; //레거시 용도
const addBtn = document.querySelector('.add-btn');
const addInput = document.querySelector('.add-input');

addBtn.addEventListener('click', () => addMenu(data));
//add 버튼 클릭하면 addMenu 함수 콜

addInput.addEventListener('keypress', (e) => e.code === 'Enter' && addMenu(data));
//addinput에서 엔터 누르면 addMenu 함수 콜
// e.code === 'ENTER' 가 true 로 평가되어야 addMenu(data) 가 호출
// addMenu(data)를 리턴하라는 듯 보이지만, 어떤 값이 아니라 실행문이라 함수가 실행됨

-----add.js
const addInput = document.querySelector('.add-input');
```

```

const list = document.querySelector('.list');

function add(data) {
 const inputValue = addInput.value;
 const index = data.length;

 const li = document.createElement('li');
 li.classList.add('item');
 li.innerHTML = `${index + 1}${inputValue}<button class="del-btn" data-index="${index}">x</button>`;
 list.append(li);

 data.push(inputValue);

 addInput.value = '';
 addInput.focus();
}

function emptyAlert() {
 alert('고민되는 메뉴를 입력해 주세요.');
 addInput.focus();
}

function maxAlert() {
 alert('메뉴는 최대 5개까지만 고민할 수 있습니다.');
 addInput.value = '';
}

// 여기까지 data를 가져오는건 최대치 확인 위함 뿐임, 레거시 남기는 것
// 위 add 함수보면 실제 list html에 넣는 것은 inputValue가 함
export function addMenu(data) {
 const inputValue = addInput.value;

 if (inputValue === '') {
 emptyAlert();
 } else if (data.length > 4) {
 maxAlert();
 } else {
 add(data);
 }
}

```

- 이름 바꾸기

`import { 원래 변수 as 여기서 바꿔서 쓸 이름의 변수 }`

- 한꺼번에
  - `import * as printerJS from`

쓸 땐 프로퍼티 형식으로 `printerJS.title` 등으로 사용

이름 중복 걱정 x, 각 변수 함수마다 다 써줄 수 없을 때 장점

◦ `export { title as title1, print }` - 변수나 함수 모두 가능

이렇게 여기서 미리 이름 바꿀 수도 있음

모든 선언문마다 `export` 붙이기 귀찮으니까!

- `tag.js` 활용

`document.querySelector` 메소드를 각 모듈에서 매번 사용하기 번거로우니 `tag.js` 따로 하나 만들어서 거기서만 선언하고 `import` 해서 사용하자

```

-----tag.js
const addBtn = document.querySelector('.add-btn');
const addInput = document.querySelector('.add-input');
const list = document.querySelector('.list');

export {addBtn, addInput, list}

-----add.js 등 다른 많은 모듈들에서
import {list, addInput} from './tags.js'

```

- `default export`

```
export default Home;
```

```
import Home from "./routes/Home";
```

- o named export - 그 파일에서 ex 할 대상이 여러개

```
export {addBtn, addInput, list}
import {list, addInput} from './tags.js'
```

- o default export - 그 파일에서 ex 할 대상이 한개, 간결해짐 (export 여러개 가능하긴함)

```
export default title(변수)
export default 'codeit' 처럼 값 자체를 전달할수도, 선언 한번만 가능
import { default as codeit } from './~'
import codeit from '~' 으로 간단하게 쓸 수 있음 (중괄호 없이!)
```

- o 보통 둘 중 하나만 사용하긴함, 디폴트 왜쓰는지 모르겠음

```
export default rollMenu
import rollMenu from './roll.js'
```

```
// (modules.js)
import module1 from './sub-module1.js';
import module2 from './sub-module2.js';
import module3 from './sub-module3.js';

export { module1, module2, module3 };

// index.js
import { module1, module2, module3 } from 'modules.js';
```

- o 모듈 예제 (메뉴 추가, 삭제, roll) 최종본

```
//tags.js로 매번 html에 접근하기 위한 document.querySelector 생략
import { addBtn, addInput, list, rollBtn } from './tags.js';
//functions.js는 각 모듈에서 쓰이는 함수를 모듈마다 취합하고 index.js 여기서 한번에 뿌려주기 위함
//안그리면 import addMenu from 'add.js' 이거 3줄 반복해야함
import {addMenu, deleteMenu, rollMenu} from './functions.js'

const data = [];

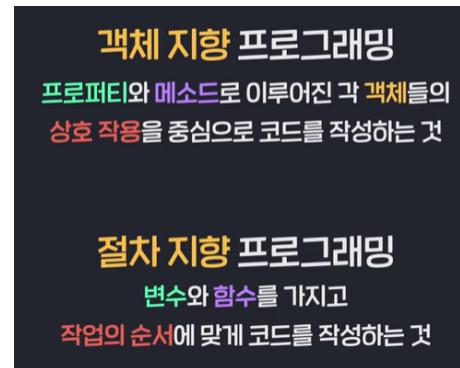
addBtn.addEventListener('click', () => addMenu(data));
addInput.addEventListener('keypress', (e) => e.code === 'Enter' && addMenu(data));
list.addEventListener('click', ({ target }) => target.tagName === 'BUTTON' && deleteMenu(data, target.dataset.index));
rollBtn.addEventListener('click', () => list.classList.contains('rolling') || rollMenu(data));
```

```
// 여기에 코드를 작성해 주세요.
import addMenu from './functions/add.js'
import deleteMenu from './functions/delete.js'
import rollMenu from './functions/roll.js'

export {addMenu, deleteMenu, rollMenu};
```

```
export default addMenu;
```

## ▼ 객체 지향 자바스크립트



객체 = 변수 & 함수 묶는 개념

- object-literal

```
const user1 = {
 email: 'chris123@google.com',
 birthDate: '1992-03-21',
 buy(item) {
 console.log(`${this.email} buys ${item.name}`);
 },
};
```

- 객체를 만들어내는 함수, (위처럼 비효율적으로 말고)

- factory function

```
function createUser (name, birth) {
 const user = { //이 블록은 객체 그 자체, 선언
 name, //name : name 을 간소화 한 것, 동일하므로
 birth,
 buySome(item) {
 console.log(`${this.name} born ${this.birth} buy ${item.names}`)
 }
 //함수 역시 간소화 표현, this는 이걸 콜한 객체(user)를 가리킴
 }
 return user; //이 선언된 객체를 리턴하겠다
}

const item = {
 names : 'apple',
 price : 15
}

//함수 콜해서 리턴값으로 객체 받음
const user1 = createUser('John', 2000)
const user2 = createUser('Park', 2001)

console.log(user1.name)
user1.buySome(item)
```

- constructor func

```
function User(name, birth) { //첫 문자가 대문자 = 관습
 this.name = name, // 객체 형식이 아니므로 =로 할당, this를 꼭붙이네?
 this.birth = birth,
 this.buySome = function (item) {
 console.log(`${this.name} born ${this.birth} buy ${item.names}`)
 }
}

const item = {
 names : 'apple',
 price : 15
}

const user1 = new User("John", 2000)
const user2 = new User("park", 2010) //new를 붙인다
```

- class

- constructor 메소드는 생성자, 객체 생성시 실행됨
- 프로퍼티와 메소드를 분리, 프로퍼티는 컨스트럭터 안에 선언
- 객체 내 getfullname : function() {}  
getfullname() {}로 축약하기도함
- class 문법에서는 딱히 프로퍼티 선언문 const name이나 name:name 등이 없고 this.name = name 으로 변수 선언이 되는 듯

```

class User {
 constructor (name, birth) {
 this.name = name;
 this.birth = birth;
 }

 buySome(item){
 console.log(` ${this.name} born ${this.birth} buy ${item.names}`)
 }
 //buySome = function(item) 도 됨
}

const item = {
 names : 'apple',
 price : 15
}

const user1 = new User("John", 2000)
const user2 = new User("park", 2010) //new를 붙인다

console.log(user1.name)
user1.buySome(it)

```

#### 객체지향 프로그래밍 핵심 개념 4가지 기둥

- 추상화
  - class 만드는 과정
  - 객체의 속성=프로퍼티, 행동=메소드 표현할 때 적절한 이름 짓기
- 캡슐화
  - 객체 접근 시 프로퍼티나 메소드 직접 값 설정 못하게하는 것, 이상한 값 할당을 막는 용도 = 값 유효성 검사
  - setter 메소드
    - set ~ = 프로퍼티 값 설정 시 불리는 함수
    - \_email 은 숨기고싶은 프로퍼티, 그냥 email은 함수 이름이 되는 것
  - getter 메소드 = 프로퍼티 값을 읽는 용도
    - \_email로 접근해야 하는 것을 email로 접근할 수 있게 하는 함수

```

class User {
 constructor(email, birthdate) {
 this.email = email;
 this.birthdate = birthdate;
 }

 buy(item) {
 console.log(` ${this.email} buys ${item.name}`);
 }

 get email() {
 return this._email; //this. 해줘야 이 객체의 email임을 알게됨
 }

 set email(address) { //address에 할당한 값 들어감
 if (address.includes('@')) {
 this._email = address;
 } else {
 throw new Error('invalid email address'); //의도적 에러 호출
 }
 }
}

const user1 = new User('chris123@google.com', '1992-03-21');

```

```
user1.email = 'newChris123@google.com'; //여기서 set email(값) 실행 된 것임
console.log(user1.email); //여기선 get 함수가 불리겠지, 그러니까 바로 email 쓸 수 있는거임
```

- user1.\_email = 'chris robert';

이런 식으로 여전히 직접 접근할 수는 있음 = 클로저 함수로 완벽한 캡슐화 한다~라는 것만

- 상속

- 부모의 프로퍼티, 메소드 물려받음 = 공통된 중복되는건 없애야지!
- class PremiumUser extends User
- 자식 컨스트럭터 안에서 super(물려받는 프로퍼티)로 호출해야
  - 자식 클래스에서 부모 클래스의 생성자 함수나 일반 메소드에 접근하려면!

```
class SavingsAccount extends BankAccount{
 constructor(name, money, years=0) {
 super(name,money); // this.holder = name; 부모에서 이렇게 선언했어도 super(name)이 맞음
 this.years = years //새로운 프로퍼티 선언, 기본값=0
 }

 addInterest(rate) {
 this.balance *= (1 + (rate * this.years));
 }
 //메소드는 아무것도 super 같은거 안해도 다 물려받나봄
}
```

- 다형성

- = 하나의 변수가 다양한 종류의 클래스로 만든 여러 객체를 가리킬 수 있음
- 자식에서 같은 메소드의 수정이 필요하다면 오버라이딩 (덮어쓰기)
  - = 아예 선언을 다시하는 것 가능
- 하지만 같은 부분은 남기고 다른 일부분만 수정하려면

```
transfer(money, anotherAccount) {
 super.transfer(money, anotherAccount); //이건 공통된 부분 갖고오기 위함.
 this.balance -= money * 0.005; //부모 transfer 과 다른 점
}
```

```
class BankAccount {
 constructor(name, money) {
 this.holder = name;
 this.balance = money;
 }

 get balance() {
 return this._balance;
 }

 set balance(money) {
 if (money >= 0) {
 this._balance = money;
 } else {
 console.log('Not valid');
 }
 }

 deposit(money) {
 this.balance += money;
 }

 withdraw(money) {
 if (this.balance - money < 0) {
 console.log('Insufficient balance');
 } else {
 this.balance -= money;
 }
 }

 transfer(money, anotherAccount) { //이 블록 내 기능이 자식 transfer 함수에서도 가능
 const account = anotherAccount;
 }
}
```

```

 if (this.balance - money < 0) {
 console.log('Insufficient balance');
 } else {
 this.balance -= money;
 account.balance += money;
 }
 }

class SavingsAccount extends BankAccount {
 constructor(name, money) {
 super(name, money);
 this.years = 0;
 }

 addInterest(rate) {
 this.balance *= (1 + (rate * this.years));
 }

 transfer(money, anotherAccount) {
 super.transfer(money, anotherAccount); //이건 공통된 부분 갖고오기 위함.
 this.balance -= money * 0.005; //부모 transfer 과 다른 점
 }
}

class DonationAccount extends BankAccount {
 constructor(name, money, rate) {
 super(name, money);
 this.rate = rate;
 }

 donate(rate) {
 this.balance *= (1 - this.rate);
 }

 transfer(money, anotherAccount) {
 super.transfer(money, anotherAccount);
 this.balance -= money * 0.002;
 }
}

const ba1 = new BankAccount('Tom', 20000000);
const sa1 = new SavingsAccount('Jerry', 10000000);
const da1 = new DonationAccount('Kate', 30000000);
const sa2 = new SavingsAccount('Alice', 9000000);

const accountForVacation = new BankAccount('Vacation', 0);

const people = [ba1, sa1, da1, sa2] //한번에 가능
for (let person of people){
 person.transfer(800000, accountForVacation);
 //모두 부모 메소드를 super 했고 & transfer로 함수이름이 통일되었고
 //다형성 = 같은 이름 transfer가 bank계좌의 객체를 가리키기도, saving계좌의 객체를 가리키기도
}

console.log(ba1.balance);
console.log(sa1.balance);
console.log(da1.balance);
console.log(sa2.balance);
console.log(accountForVacation.balance);

```

- instanceof 연산자

어느 클래스로부터 만들어진 객체인지 알려면

- 객체 instanceof 클래스이름
  - 자식 class로 만들어졌어도 부모 값은 true
- static 프로퍼티, 메소드
  - 클래스에 직접적으로 달려있는
  - 객체가 아닌 클래스 자체로 접근
  - 클래스가 객체를 생성하는 용도가 아닐수도 있음  
Math.PI , Date.now()

```

class Math {
 static PI = 3.14;

 static getCircleArea(radius) {
 return Math.PI * radius * radius;
 }
}

console.log(Math.PI);
console.log(Math.getCircleArea(5));

```

- 파일관리

- main.js

```

import User from './User';
import PremiumUser from './PremiumUser';

const user3 = new User('brian@google.com', '20051125');
const pUser1 = new PremiumUser('niceguy@google.com', '19891207', 3);

```

- PremiumUser.js

```

import User from './User';

class PremiumUser extends User {

 export default PremiumUser

```

- User.js

```

class User {

 export default User;

```

▼ User & premiumUser 최종 코드

```

class User {
 constructor(email, birthdate) {
 this.email = email;
 this.birthdate = birthdate;
 }

 buy(item) {
 console.log(` ${this.email} buys ${item.name}`);
 }
}

class PremiumUser extends User {
 constructor(email, birthdate, level, point) {
 super(email, birthdate);
 this.level = level;
 this.point = point;
 }

 buy(item) {
 console.log(` ${this.email} buys ${item.name}`);
 this.point += item.price * 0.05;
 }

 streamMusicForFree() {
 console.log(`Free music streaming for ${this.email}`);
 }
}

const item = {
 name: '스웨터',
 price: 30000,
};

const user1 = new User('chris123@google.com', '19920321');
const user2 = new User('rachel@google.com', '19880516');
const user3 = new User('brian@google.com', '20051125');
const pUser1 = new PremiumUser('niceguy@google.com', '19891207', 3);

```

```

const pUser2 = new PremiumUser('helloMike@google.com', '19900915', 2);
const pUser3 = new PremiumUser('aliceKim@google.com', '20010722', 5);

const users = [user1, pUser1, user2, pUser2, user3, pUser3];

users.forEach((user) => {
 user.buy(item);
});

```

### ▼ 자바스크립트 웹 개발

- 웹 브라우저와 서버 사이의 통신!
- 개발자 도구 단축키 = **ctrl + shift + i**
  - 코드를 여러 줄 연달아 작성하려면 Enter 말고 **Shift + Enter**
  - 아무런 값도 리턴하지 않는 경우에는 `undefined`를 리턴한 것으로 간주
  - network 칸에 req resp 내역 볼 수 있음
- 그러기 위해서 웹 브라우저의 js 코드 - json, ajax, promise, fetch, await/async 배우게 됨
- `fetch, response`로 요청해서 받은 promise ~웹 브라우저가 서버로부터 받은 응답인 html js 코드, 이걸 웹 브라우저가 해석해서 우리가 보이는 사이트
- `fetch` 함수로 request 하여 response 받게됨

```

fetch('https://google.com')
 .then((response) => (response.text())) //response라는 이름으로 객체를 받아서 그의 text()라는 함수를 리턴한다
 .then((result) => {console.log(result)})

```

- `fetch` 함수는 **Promise** 객체를 리턴하는데, 이 객체의 `then` 메소드로 콜백을 등록할 수 있다
  - 콜백함수는 당장 실행되는 것이 아니라 추후에 특정 조건(여기서는 서버의 리스폰스가 도착했을 때)이 만족되었을 때 실행되는 함수
  - 2번째 `then` 메소드는 위 `then` 메소드가 실행 완료된 후에 실행
  - 이전 콜백의 리턴값이 다음 콜백의 파라미터로 넘어감
  - `response` 객체의 `text()` 리턴값이 `response`의 실제 내용이라 이렇게 2번 실행함 (그냥 `response`는 부가 정보들 투성 이)
- WEB이란 : 수많은 웹 페이지의 가상의 연결망, 서로 연결되어있음
- URL : 웹에 존재하는 특정 데이터 나타내는 문자열
  - `http ~`
  - **호스트** = 전세계 서버 중 하나의 서버
  - **경로 (path)** = 서버 내 데이터 중 원하는 데이터 특정
  - **?쿼리** = 있을수도 없을수도, 데이터에 관한 세부적인 요구사항
  - 웹 페이지에서 버튼을 클릭하면 지금 보이는 것 같은 a 태그의 href 속성에 적힌 URL 주소로 웹 브라우저가 알아서 리퀘스트를 보내서 리스폰스를 받아 새로운 웹 페이지를 로드
    - 먼저 호스트 (도메인 네임)을 보고 어떤 서버와 통신해야 하는지 식별
    - 그 서버에 path와 그 이후의 것을 request에 담아 보냄
    - 서버는 그것들에 해당 데이터를 찾아 response에 담아 보냄
    - 웹 브라우저가 받아 그 데이터나 html js 코드에 해당 웹 페이지 띄움
- https = 스킴 = 프로토콜의 이름
  - **프로토콜** : 두 주체가 지켜야 하는 통신 규약, 이것에 맞게 req resp 주고 받음
  - http에서 보안성을 더한 https 가 대표적 프로토콜
- 보통 브라우저가 하나의 페이지를 그릴 때는 첫 리스폰스의 내용 안에서 또다시 요구되는, 여러 가지 다른 것들을 구하기 위해 다시 여러 개의 리퀘스트를 보내

- JSON
  - html 코드가 아닌 정보가 담긴 response도 있다
  - 이 형태를 받을 때의 포맷 = javascript object notation
  - 자바스크립트 문법과 비슷한, 배열로 감싸고 중괄호 안에 객체 표기
    - 프로퍼티는 무조건 "", 값도 문자열이면 무조건 ""
    - 프로퍼티의 값으로 undef, nan, infi 불가
    - 주석 추가 불가
  - json ⇒ 객체 변환
    - json의 typeof 는 string ㅠㅠ 접근 어려움
    - JSON.parse(result)

스트링 타입의 json을 자바스크립트 배열(객체)의 타입으로 변환

```
// 모든 json 데이터 다 출력
// fetch('https://learn.codeit.kr/api/topics')
// .then((response) => response.text())
// .then((result) => { console.log(result) })

//초급만 걸러져서 나옴
fetch('https://learn.codeit.kr/api/topics')
 .then((response) => response.text())
 .then((result) => {
 const topics = JSON.parse(result) //filter 함수 쓰기 위해 배열 형태로 바꿈
 const beginnerLevelTopics = topics.filter((topic) => topic.difficulty === '초급');
 console.log(beginnerLevelTopics);
 });
});
```

- 메소드
  - request의 종류 CRUD
    - 데이터 조회 read = get
    - 데이터 추가 create = post
    - 데이터 수정 update = put
    - 데이터 삭제 delete = delete
  - request 구조
    - head
      - = req의 부가정보
      - 여러 header
 

```
method: GET
path: /users
user-agent: req 보낸 브라우저와 운영체제 정보 등
```
    - body
      - = 실제 데이터 담는 정보
      - post, put 만 필요 / get, delete는 필요없는 부분
- req 보내기
  - 옵션 객체 - fetch 두번째 파라미터
 

```
{ method: ~ , body: ~ }
```
  - 자바스크립트 객체를 json 외부로 내보내려면 스트링으로 바꿔JSON.stringify(객체)
  - get
    - 특정 조회 - url 뒤에 /3 추가

- 옵션 객체 안적으면 디폴트로 get임
- url로 접속하는 거 자체가 get req 보내는것과 동일

```
fetch("https://learn.codeit.kr/api/members/3") //특정 멤버 조회
 .then((response) => (response.text()))
 .then((result) => (console.log(result)));
```

- post

```
fetch("https://learn.codeit.kr/api/members", { //두번째 파라미터는 객체 형식
 method: 'POST', //메소드 특정
 body: JSON.stringify(newMember) //json 형태로 추가해줘야 하므로 string화
})
 .then((response) => (response.text()))
 .then((result) => (console.log(result)));

const newMember = {
 name: 'a',
 email: 'd',
 department: 'c',
}
```

- put

```
fetch("https://learn.codeit.kr/api/members/3", { //수정할 멤버의 인덱스 적어줌!
 method: 'PUT', //
 body: JSON.stringify(Member)
})
 .then((response) => (response.text()))
 .then((result) => (console.log(result)));

const Member = { //수정할 프로퍼티만 수정하고 나머진 그대로
 name: 'aaaaaaaaaa', //
 email: 'd',
 department: 'c',
}
```

- delete

```
fetch("https://learn.codeit.kr/api/members/3", { //삭제할 멤버의 인덱스 적어줌!
 method: 'DELETE',
 //body 필요 없음
})
 .then((response) => (response.text()))
 .then((result) => (console.log(result)));
```

- 실습

```
// 새 직원 정보는 원하는 대로 작성하세요
const newMember = {
 name: "조수하",
 email: "dlrhdcjs@naver.com",
 department: "sw"
};

fetch('https://learn.codeit.kr/api/members', {
 method: 'POST',
 body: JSON.stringify(newMember)
}) //추가 한다
 .then(() => { //추가 완료되면 실행될 콜백함수
 fetch('https://learn.codeit.kr/api/members') //조회겠지
 .then((response) => response.text())
 .then((result) => { //result는 전체 멤버 json 형식
 const members = JSON.parse(result) //json -> 객체
 console.log(members[members.length - 1]); //객체의 마지막(추가 시 맨 마지막에 추가되더라)
 });
 });
});
```

- Web API 설계

- 우리가 어떤 리퀘스트를 보냈을 때 무슨 리스폰스를 받는지 어떻게 설계? - fe,be가 모여서 회의
- Web API가 잘 설계되었는지에 관한 기준으로는 보통 **REST API**  
라는 기준이 사용, 그 중 우리가 지킬 수 있을 만한 것들
  - URL은 리소스를 나타내는 용도로만 사용하고 (동사 사용x), 리소스에 대한 처리는 메소드로 표현해야 한다
  - URL에서 도큐먼트는 단수 명사로, 컬렉션은 복수 명사로 표현

| 제목     | /members              | /members/3  |
|--------|-----------------------|-------------|
| GET    | 전체 직원 정보 조회           | 3번 직원 정보 조회 |
| POST   | 새 직원 정보 추가            | X           |
| PUT    | 전체 직원 정보 수정(잘 쓰이지 않음) | 3번 직원 정보 갱신 |
| DELETE | 전체 직원 정보 삭제(잘 쓰이지 않음) | 3번 직원 정보 삭제 |

\* post 추가 시 추가될 직원 정보가 어떤 id 값을 할당받을지 알 수도 없기 때문에 /members로만

#### ▼ web api, rest api 구체적 설명

우리는 이제 웹 브라우저가 리퀘스트를 보낼 때

- (1) 어느 **URL**로 리퀘스트를 보내는지
- (2) 무슨 **메소드**(GET, POST, PUT, DELETE 등)가 그 헤드에 설정되어있는지가  
중요하다는 것을 배웠습니다.

그런데 우리가 어떤 리퀘스트를 보냈을 때, 무슨 리스폰스를 받는지는 어떻게 설계되는 걸까요? 개발자들이 실제로 개발을 할 때 이 부분을 어떻게 만들고 있는지 이번 노트에서 배워보겠습니다.

## 1. Web API

우가 어떤 리퀘스트를 보냈을 때, 무슨 리스폰스를 받는지는 모두 그 서비스를 만드는 개발자들이 정하는 부분입니다. 잠깐 실제 개발 현장에서 일어나는 이야기를 해볼게요. 개발자에는 크게 두 가지 종류가 있습니다. 첫 번째는 사용자가 직접 서비스 화면을 보는 웹 페이지나 앱 등을 만드는 프론트엔드(Front-end) 개발자, 두 번째는 웹 브라우저나 앱이 보내는 리퀘스트를 받아서 적절한 처리를 한 후 리스폰스를 주는 서버의 프로그램을 만드는 백엔드(Back-end) 개발자, 이 두 가지인데요.

하나의 서비스를 만들 때는 프론트엔드 개발자들과 백엔드 개발자들이 모여 '프론트엔드에서 이 URL로 이렇게 생긴 리퀘스트를 보내면, 백엔드에서 이런 처리를 하고 이런 리스폰스를 보내주는 것으로 정합시다'와 같은 논의를 하고, 이런 내용들을 정리한 후에 개발을 시작합니다.

이것을 'Web API 설계'라고 하는데요. API란 Application Programming Interface의 약자로, 원래는 '개발할 때 사용할 수 있도록 특정 라이브러리나 플랫폼 등이 제공하는 데이터나 함수 등'을 의미합니다. 웹 개발에서는 어느 URL로 어떤 리퀘스트를 보냈을 때, 무슨 처리가 수행되고 어떤 리스폰스가 오는지에 관해 미리 정해진 규격을 **Web API**라고도 하는데요.

Web API를 설계한다는 것은 서비스에서 사용될 모든 URL들을 나열하고, 각각의 URL에 관한 예상 리퀘스트와 리스폰스의 내용을 정리한다는 뜻입니다. 예를 들어, 이전 영상에서 사용한 학습용 URL(<https://learn.codeit.kr/api/members>)에서 직원 정보 추가 기능을 설계한다면 다음과 같이 할 수 있는 겁니다.

```

...
3. 직원 정보 추가
https://learn.codeit.kr/api/members

(1) Request
- Head
Method : POST
...

- Body
{
 "name": "Jerry",
 "email": "jerry@codeitshopping.kr",
 "department": "engineering",
}
...

(2) Response
Success인 경우 :
- Head
...

```

```

- Body
{
 "id": "[부여된 고유 식별자 값]",
 "name": "Jerry",
 "email": "jerry@codeshopping.kr"
 "department": "engineering",
}
Fail인 경우 :
...

```

이렇게 해당 서비스에서 제공되는 각 URL에, 어떤 리퀘스트를 보내면, 서버는 어떤 리스폰스를 보내야 하는지를 일일이 설계하는 것이 Web API 설계인 겁니다. 물론 실무에서는 지금 보이는 예시보다 훨씬 체계적이고 단정한 방식으로, 상용 툴 등을 사용해서 정리하지만 일단은 이해 차원에서 보여드렸습니다. 이런 식으로 Web API가 설계되고 나면, 그때 프론트엔드/백엔드 개발자들이 해당 설계에 맞게 각자 코드를 작성하기 시작하는 겁니다. 물론 설계와 개발이 동시에 진행되기도 하고, 설계 내용이 중간에 수정되기도 합니다.

오늘날 많은 회사 내의 개발팀은 이런 식으로 Web API를 설계하고 웹 서비스를 만듭니다. 그런데 문제가 하나 있습니다. 그건 바로 **Web API는 어떻게 설계해도 동작하는 데는 아무런 지장이 없다는 문제입니다.**

이전 영상들에서 저는 직원 정보를 추가하기 위해

- (1) 'https://learn.codeit.kr/api/members' URL로
- (2) 리퀘스트의 헤드에 POST 메소드를 설정하고,
- (3) 리퀘스트의 바디에 새 직원 정보를 넣어서 보내면 된다

는 내용의 설계를 했습니다.

그런데 어떤 회사는 같은 기능을 이런 식으로 설계할 수도 있습니다.

- (1) 'https://learn.codeit.kr/api/members' URL로
- (2) 리퀘스트의 헤더에 GET 메소드를 설정하고,
- (3) 리퀘스트의 바디에 새 직원 정보를 넣어서 보내면 된다

어느 방식으로 설계해도 서비스가 동작하는 데는 아무런 문제가 없습니다. 하지만 기능적으로 아무런 문제가 없다고 해도 Web API를 아무렇게나 설계해도 되는 것은 아닙니다. 사실 Web API가 잘 설계되었는지에 관한 기준으로는 보통 **REST API**라는 기준이 사용되고 있는데요. 많은 개발자들이 Web API를 개발할 때 이 REST API를 준수하기 위해 노력하고 있습니다. 이게 뭔지 한번 살펴봅시다.

## 2. REST API 이야기

**REST API**는 오늘날 많은 웹 개발자들이 Web API 설계를 할 때, 준수하기 위해 노력하는 일종의 가이드라인입니다. REST API를 이해하기 위해서는 일단 **REST architecture**가 무엇인지부터 알아야 하는데요. 일단 REST architecture에 대해 설명하겠습니다.

REST architecture란 미국의 컴퓨터 과학자인 Roy Fielding이 본인의 박사 논문 'Architectural Styles and the Design of Network-based Software Architectures'에서 제시한 개념인데요. 그는 웹이 갖추어야 할 이상적인 아키텍처(구조)로 REST architecture라는 개념을 제시했습니다. 여기서 REST는 **Representational State Transfer**(표현적인 상태 이전)의 줄임말로, 해석하면 '표현적인, 상태 이전'이라는 뜻입니다. 이게 무슨 말일까요? 이 용어는 Roy Fielding이 고안한 용어인데요. 지금 여러분이 웹 서핑을 할 때를 생각해보세요. 만약 웹을 하나의 거대한 컴퓨터 프로그램이라고 생각한다면, 각각의 웹 페이지는 그 프로그램의 내부 상태를 나타낸다고 할 수 있습니다. 그렇다면 우리가 웹 페이지들을 계속 옮겨 다니면서 보게 되는 내용은, 웹이라는 프로그램의 매번 새로운 상태를 나타내는 표현이라고 할 수 있는데요. 그래서 이것을 '표현적인, 상태 이전'이라고 하는 겁니다. 조금 추상적인 느낌이지만 이해는 되시죠?

그럼 REST architecture가 되기 위한 조건에는 어떤 것들이 있을까요? 다음과 같은 6가지 기준을 충족하면 REST architecture로 인정됩니다.

1. Client-Server
2. Stateless
3. Cache
4. Uniform Interface
5. Layered System
6. Code on Demand

각 기준에 대해 간략하게 설명해보자면 REST architecture는,

1. **(Client-Server)** Client-Server 구조를 통해 양측의 관심사를 분리해야 합니다. 현재 토픽에서는 웹 브라우저가 실행되고 있는 컴퓨터가 Client, 서비스를 제공하는 컴퓨터가 Server에 해당하는데요. 이렇게 분리를 해놓으면 Client 측은 사용자에게 어떻게 하면 더 좋은 화면을 보여줄지, 다양한 기기에 어떻게 적절하게 대처해야 할지 등의 문제에 집중할 수 있고, Server 측은 서비스에 적합한 구조, 확장 가능한 구조를 어떻게 구축할 것인지 등의 문제에 집중할 수 있습니다. 이렇게 각자가 서로를 신경쓰지 않고 독립적으로 운영될 수 있는 겁니다.
2. **(Stateless)** Client가 보낸 각 리퀘스트에 관해서 Server는 그 어떤 맥락(context)도 저장하지 않습니다. 즉, 매 리퀘스트는 각각 독립적인 것으로 취급된다는 뜻입니다. 이 때문에 리퀘스트에는 항상 필요한 모든 정보가 담겨야합니다.
3. **(Cache)** Cache를 활용해서 네트워크 비용을 절감해야 합니다. Server는 리스폰스에, Client가 리스폰스를 재활용해도 되는지 여부(Cacheable)를 담아서 보내야 합니다.
4. **(Uniform Interface)** Client가 Server와 통신하는 인터페이스는 다음과 같은 하위 조건 4가지를 준수해야 합니다. 이 조건이 REST API와 연관이 깊은 조건입니다. 어떤 4가지 하위 조건들이 있는지 살펴봅시다.

(4-1) identification of resources : 리소스(resource)는 웹상에 존재하는 데이터를 나타내는 용어인데요. 저도 이번 노트에서는 리소스라는 용어를 사용하겠습니다. 이것은 리소스(resource)를 URI(Uniform Resource Identifier)로 식별할 수 있어야 한다는 조건입니다. URI는 URL의 상위 개념으로 일단 지금은 URL이라고 생각하셔도 큰 무리는 없습니다.

(4-2) manipulation of resources through representations : Client와 Server는 둘 다 리소스를 직접적으로 다루는 게 아니라 리소스의 '표현(representations)'을 다뤄야 합니다. 예를 들어, Server에 '오늘 날씨'(/today/weather)라는 리소스를 요청했을 때, 어떤 Client는 HTML 파일을 받을 수도 있고, 어떤 Client는 이미지 파일인 PNG 파일을 받도록 구현할 수도 있는데요. 이때 HTML 파일과 PNG 파일 같은 것들이 바로 리소스의 '표현'입니다. 즉, 동일한 리소스라도 여러 개의 표현이 있을 수 있다는 뜻입니다. 사실, 리소스는 웹에 존재하는 특정 데이터를 나타내는 추상적인 개념입니다. 실제로 우리가 다루게 되는 것은 리소스의 표현들뿐인데요. 이렇게 '리소스'와 '리소스의 표현'이라는 개념 2개를 서로 엄격하게 구분하는 것이 REST architecture의 특징입니다.

(4-3) self-descriptive messages : self-descriptive는 '자기설명적인'이라는 뜻인데요. 위에서 살펴본 2. Stateless 조건 때문에 Client는 매 리퀘스트마다 필요한 모든 정보를 담아서 전송해야 합니다. 그리고 이때 Client의 리퀘스트와 Server의 리스폰스 모두 그 자체에 있는 정보만으로 모든 것을 해석할 수 있어야 한다는 뜻입니다.

(4-4) hypermedia as the engine of application state : REST architecture는 웹이 갖추어야 할 이상적인 아키텍처라고 했죠? 이때 '웹'을 좀더 어려운 말로 풀어써 보자면 '분산 하이퍼미디어 시스템'(Distributed Hypermedia System)이라고도 할 수 있는데요. 여기서 하이퍼미디어(Hypermedia)는 하이パーテ스트(Hypertext)처럼 서로 연결된 '문서'에 국한된 것이 아니라 이미지, 소리, 영상 등까지도 모두 포함하는 더 넓은 개념의 단어입니다. 즉, 웹은 수많은 컴퓨터에 하이퍼미디어들이 분산되어 있는 형태이기 때문에, '분산 하이퍼미디어 시스템'에 해당합니다. 이 조건은 웹을 하나의 프로그램으로 간주했을 때, Server의 리스폰스에는 현재 상태에서 다른 상태로 이전할 수 있는 링크를 포함하고 있어야 한다는 조건입니다. 즉, 리스폰스에는 리소스의 표현, 각종 메타 정보들뿐만 아니라 계속 새로운 상태로 넘어갈 수 있도록 해주는 링크들도 포함되어 있어야 한다는 거죠.

자, 여기까지가 Uniform Interface의 4가지 하위 조건입니다. 사실, 오늘날 우리가 Web API를 설계할 때 위의 하위 조건들을 모두 제대로 이해하고 준수하는 것은 쉽지 않은 일인데요. 일단 아직 남은 5, 6번 조건들을 마지막으로 살펴보고, 4번에 관해 그나마 우리가 실천할 수 있는 규칙들을 아래에서 살펴봅시다.

1. **(Layered System)** Client와 Server 사이에는 프록시(proxy), 게이트웨이(gateway)와 같은 중간 매개 요소를 두고, 보안, 로드 밸런싱 등을 수행할 수 있어야 합니다. 이를 통해 Client와 Server 사이에는 계층형 층(hierarchical layers)들이 형성됩니다.
2. **(Code-On-Demand)** Client는 받아서 바로 실행할 수 있는 applet이나 script 파일을 Server로부터 받을 수 있어야 합니다. 이 조건은 Optional한 조건으로 REST architecture가 되기 위해 이 조건이 반드시 만족될 필요는 없습니다.

자, 이때까지 REST architecture가 되기 위해 충족해야 하는 조건들을 배웠는데요. 이해가 잘 되는 것도 있고 조금 어려운 것도 있죠? 사실 이 내용은 다소 이론적이기도 하고, 웹에 대해 좀 더 많이 공부해야 이해할 수 있는 것들도 있기 때문에 일단은 그냥 넘어가셔도 괜찮습니다.

하지만 기억해야 할 사실은, REST API는 바로 이런 REST architecture에 부합하는 API를 의미한다는 사실입니다. 참고로 이런 REST API를 사용하는 웹 서비스를 RESTful 서비스라고 합니다. 그렇다면 구체적으로 어떤 식으로 Web API를 설계해야 REST API가 될 수 있는 걸까요? 사실 Roy Fielding의 논문에는 이것에 관한 구체적이고 실천적인 내용들은 제시되어 있지 않습니다. 하지만 많은 개발자들의 경험과 논의를 통해 형성된 사실상의(de facto) 규칙들이 존재하는데요.

우리는 그중에서도 조건 4. Uniform Interface의 하위 조건인 (4-1) identification of resources에 관해서 특히 개발자들이 강조하는 규칙, 2가지만 배워보겠습니다.

## (1) URL은 리소스를 나타내기 위해서만 사용하고, 리소스에 대한 처리는 메소드로 표현해야 합니다.

이 규칙은 조금 다르게 설명하자면, URL에서 리소스에 대한 처리를 드러내면 안 된다는 규칙인데요. 이게 무슨 말인지 1. Web API 부분에서 마지막에 언급했던 예시를 통해 이해해보겠습니다.

예를 들어, 새 직원 정보를 추가하기 위해서

- (1) 'https://learn.codeit.kr/api/members' URL로
- (2) 리퀘스트의 헤드에 POST 메소드를 설정하고,
- (3) 리퀘스트의 바디에 새 직원 정보를 넣어서 보내면 된다

고 하는 경우는, URL은 리소스만 나타내고, 리소스에 대한 처리(리소스 추가)는 메소드 값인 POST로 나타냈기 때문에 이 규칙을 준수한 것입니다.

하지만

- (1) 'https://learn.codeit.kr/api/members/add' URL로
- (2) 리퀘스트의 헤더에 GET 메소드를 설정하고,
- (3) 리퀘스트의 바디에 새 직원 정보를 넣어서 보내면 된다

고 하는 이 경우는 URL에서 리소스뿐만 아니라, 해당 리소스에 대한 처리(add, 추가하다)까지도 나타내고 있습니다. 그리고 정작 메소드 값으로는 리소스 추가가 아닌 리소스 조회를 의미하는 GET을 설정했기 때문에 이 규칙을 어긴 것입니다.

URL은 리소스를 나타내는 용도로만 사용하고, 리소스에 대한 처리는 메소드로 표현해야 한다는 사실, 꼭 기억하세요!

## (2) 도큐먼트는 단수 명사로, 컬렉션은 복수 명사로 표시합니다.

또 다른 규칙 하나를 살펴볼까요? 이 규칙은 URL로 리소스를 나타내는 방식에 관한 규칙인데요. URL에서는

- <https://www.soccer.com/europe/teams/manchester-united/players/pogba>

이런 식으로 path 부분에서 특정 리소스(pogba, 축구 선수 포그바의 정보)를 나타낼 때 슬래시(/)를 사용해서 계층적인 형태로 나타냅니다. 지금 위 URL의 path 부분을 보면 '유럽의', '축구팀들 중에서', '맨체스터 유나이티드 팀의', '선수들 중에서', '포그바'라는 선수의 정보를 의미하는 리소스라는 걸 한눈에 알 수 있는데요. 이렇게 계층적 관계를 잘 나타내면, URL만으로 무슨 리소스를 의미하는지를 누구나 쉽게 이해할 수 있습니다. Web API를 설계할 때는 이렇게 가독성 좋고, 이해하기 쉬운 URL을 설계해야 하는데요. 그런데 이때 지켜야 할 규칙이 있습니다.

사실 리소스는 그 특징에 따라 여러 종류로 나눠볼 수 있습니다. 이 중에서 우리는 '컬렉션(collection)'과 '도큐먼트(document)'를 배울 건데요. 보통 우리가 하나의 객체로 표현할 수 있는 리소스를 '도큐먼트'라고 합니다. 그리고 여러 개의 '도큐먼트'를 담을 수 있는 리소스를 '컬렉션'이라고 하는데요. 쉽게 비유하자면, 도큐먼트는 하나의 '파일', 컬렉션은 여러 '파일'들을 담을 수 있는 하나의 '디렉토리'에 해당하는 개념입니다.

그리고 이에 관한 규칙은 바로, **URL에서 '도큐먼트'를 나타낼 때는 단수형 명사를, '컬렉션'을 나타낼 때는 복수형 명사를 사용해야 한다는 규칙입니다.**

지금 위 URL에서 europe, manchester-united, pogba가 '도큐먼트'에 해당하고, teams, players가 '컬렉션'에 해당합니다. 도큐먼트는 단수 명사로, 컬렉션은 복수 명사로 표현한 것이 잘 보이죠?

이 규칙을 잠깐 이전 영상의 내용과 연관 지어 생각해볼까요? 예를 들어, 제가

- 전체 직원 정보 조회 - GET
- 새 직원 정보 추가 - POST

이 작업들을 수행하기 위해 사용했던 'https://learn.codeit.kr/api/members' URL에서도

직원 전체를 의미하는 members는 이렇게 복수 명사를 사용했다는 것을 알 수 있습니다. members는 member들을 담을 수 있는 컬렉션에 해당하는 개념이기 때문입니다.

그리고 제가

- 특정 직원 정보 조회 - GET
- 기존 직원 정보 수정 - PUT
- 기존 직원 정보 삭제 - DELETE

이 작업들을 수행하기 위해 사용했던 https://learn.codeit.kr/api/members/3 URL에서는

도큐먼트를 나타내기 위해 단수 명사 대신 직원 고유 식별자인 id 값을 썼는데요. 이렇게 숫자를 쓰는 경우에는 단복수 문제가 없겠죠?

'도큐먼트', '컬렉션' 개념을 우리가 배운 메소드 종류와 연결해서 모든 경우의 수를 생각해보면 다음과 같습니다.

| 제목  | /members    | /members/3  |
|-----|-------------|-------------|
| GET | 전체 직원 정보 조회 | 3번 직원 정보 조회 |

|        |                       |             |
|--------|-----------------------|-------------|
| POST   | 새 직원 정보 추가            | X           |
| PUT    | 전체 직원 정보 수정(잘 쓰이지 않음) | 3번 직원 정보 갱신 |
| DELETE | 전체 직원 정보 삭제(잘 쓰이지 않음) | 3번 직원 정보 삭제 |

지금 표에서 보이는 것처럼, 전체 직원 정보를 대상으로 PUT 리퀘스트 또는 DELETE 리퀘스트를 보내는 것은 전체 직원 정보를 모두 수정 또는 모두 삭제한다는 뜻이기 때문에 사실상 잘 쓰이지 않습니다. 위험한 동작이기 때문에 애초에 Web API 설계에 반영하지도 않고, 서버에서 허용하지 않을 때가 일반적입니다.

그리고 또 여기서 주목할 점은 **POST 리퀘스트를 보낼 때, 컬렉션(members) 타입의 리소스를 대상으로 작업을 수행한다는 점입니다.** 이 부분이 조금 헷갈릴 수도 있는데요. POST 리퀘스트를 보낼 때는 우리가 전체 직원 정보를 의미하는 컬렉션에 하나의 직원 정보(하나의 도큐먼트)를 추가하는 것이기 때문에 URL로는 컬렉션까지만 /members 이렇게 표현해줘야 합니다. 따라서 /members/3 이렇게 특정 도큐먼트를 나타내는 URL에 POST 리퀘스트를 보내는 것은 문맥상 맞지 않는 표현입니다. 그리고 지금 같은 경우는 추가될 직원 정보가 어떤 id 값을 할당받을지 알 수도 없기 때문에 애초에 /members/[id]에 id 값을 지정한다는 것도 불가능하죠.

이 도큐먼트와 컬렉션 개념을 잘 기억하고 있으면 나중에 URL에서 단수 명사를 써야 할지, 복수 명사를 써야 할지 고민이 될 때 답을 얻을 수 있을 겁니다.

자, 이때까지 REST API의 조건 중 하나인 **4. Uniform Interface**을 좀 더 잘 지키기 위해 개발자들이 강조하는 규칙 2가지를 배웠습니다. 하지만 이것만으로 Web API를 REST API로 설계할 수 있는 것은 아닙니다. 여전히 만족시켜야 하는 다른 조건들도 있기 때문이죠. 나머지 조건들을 어떻게 지킬 수 있는지에 관한 내용은 난이도 및 분량 관계상 생략하겠습니다. 나머지 조건들을 어떻게 준수하는지는 여러분이 웹에 좀더 익숙해지고 나서 나중에 더 찾아보시는 걸 추천합니다.

REST API는 개발자들이 Web API를 설계할 때 굉장히 중요하게 고려하는 가이드라인이기는 하지만, 앞서 제시한 6가지 조건을 모두 만족시켜가면서까지 100% 준수해야 할 필요성이 있는지에 관해서는 의견이 많습니다. 그래도 REST API는 웹 개발자의 주요 단골 면접 주제니까 관심이 있는 분은 이번 노트의 내용을 다시 자세히 읽고 필요한 내용을 더 찾아보세요.

- response 객체의 text 메소드 대신 json이라는 메소드를 호출하면, 리스트의 내용이 JSON 데이터에 해당하는 경우, 바로 Deserialization까지 수행해줍니다.

- 리스트의 내용이 JSON 데이터로 미리 약속된 경우 사용, 아니라면 에러 발생됨

```
fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => response.json())
 .then((result) => { const users = result; });
 //원래 text() 받으면 JSON.parse(result)
```

- Response 구조도 비슷

- head - 부가정보
  - 상태 코드 (status code)
    - 200 = req를 서버가 정상적으로 처리됨
    - 404 = url에 해당 데이터 찾을 수 없음
- body - 실제 데이터, 주로 JSON
- response.status로 확인 가능

- content-type

- req, resp 바디에 있는 데이터가 어떤 타입인지 알려주는 Header
  - html 코드 : text/html
  - json 데이터 : application/json 등등

```
fetch('https://learn.codeit.kr/api/members', {
 method: 'POST',
 headers: { // 추가된 부분
 'Content-Type': 'application/json',
 },
},
```

```
 body: JSON.stringify(newMember),
 })
```

- 여러 데이터 타입 (content-type)

- json
- xml
  - 'application/xml'
  - "name":"Michael Kim" 이걸 아래와 같이 표현
    - <name>Michael Kim</name>

- form 태그에서 사용되는 타입들

HTML의 form 태그 (<form></form>)를 사용할 때 자바스크립트 코드 없이 오로지 HTML만으로도 리퀘스트를 보내는 것이 가능, 물론 자바스크립트 코드만으로도 req의 바디에 담아 전송 가능

- **application/x-www-form-urlencoded**

- id=6&name=Jason&age=34&department=engineering 형식

- **multipart/form-data**

- 여러 종류의 데이터를 하나로 합친 데이터를 의미하는 타입, 이미지, 영상 등 첨부 시 ⇒ 매우 중요
- 각 데이터형식마다 경계선 = boundary 있는 것이 특징

```
<form action="/upload" method="post" enctype="multipart/form-data">

const formData = new FormData();
formData.append('email', email.value);
formData.append('password', password.value);
formData.append('nickname', nickname.value);
formData.append('profile', image.files[0], "me.png");

fetch('https://learn.codeit.kr/api/members', {
 method: 'POST',
 body: formData,
})
.then((response) => response.text())
.then((result) => { console.log(result); });
```

- Ajax (Asynchronous JavaScript And XML)

- Ajax 통신이 아닌 - 예전방식으로 매번 새 페이지 로드

[메인 화면으로 가기](https://learn.codeit.kr/api/main)

- Ajax 통신 = 현재 페이지 그대로 유지한 채로 서버에 req 보내고 resp 받아서 일부만 변화 = 부드러운 변화 = 사용자가 느낄 수 없게

```
function getLocationInfo(latitude, longitude) {
 fetch('https://map.google.com/location/info?lat=' + latitude + '&lng=' + longitude)
 .then((response) => response.text())
 .then((result) => { /* 사용자 화면에 해당 위치 관련 정보 띄워주기 */ });
}
```

- 자바스크립트에서는 **XMLHttpRequest** 라고 하는 객체를 통해 Ajax 통신 하지만, 굳이 객체 만들어서 번거롭게 사용 x
  - fetch 함수로 가능
  - Axios 패키지
    - XMLHttpRequest 기반 더 쓰기 편하게 만듦

- GET, POST, PUT, DELETE 이외의 메소드들

- PATCH

- { "age": 30 } 넣으면 age만 수정되고 나머지는 그대로
  - 기존 데이터의 일부 수정
  - PUT은 아예 age:30 으로 덮어쓰여짐
- HEAD
  - GET과 같지만 resp에서 바디 부분은 제외하고, 딱 헤드 부분만 받는다는 점
  - 실제 데이터가 아니라 데이터에 관한 정보만 얻으려고 하는 상황
- HTTP, HTTPS 이외에도, FTP, SSH, TCP, UDP, IP, Ethernet 등 (각 네트워크 통신의 특정 계층임) 의 프로토콜로도 통신 가능, 참고만

---

- fetch 함수의 비동기 실행
  - then은 콜백을 등록만 한다. 실행을 하는게 아니라. 콜백 실행은 resp 받았을 때 실행이 되는 것
  - 비동기 실행 = 한 번 시작한 코드가 끝나기 전에 다음 코드로 넘어가고 나중에 콜백이 실행되고 작업이 마무리 됨

```
console.log('Start!');

fetch('https://www.google.com')
 .then((response) => response.text())
 .then((result) => { console.log(result); });

console.log('End');
```



- 다른 비동기 함수
  - setTimeout(콜백, 시간)
  - setInterval(콜백, 시간)
    - setInterval(() => { console.log('b'); }, 2000);
    - 2초 간격 계속 실행
  - addEventListener(이벤트 이름, 콜백)
  - fetch 는 이들과 달리 promise 객체를 리턴하고, Promise 객체는 파라미터에 콜백을 직접 전달하던 기존의 위 방식과는 다른 방식으로 비동기 실행을 지원

**해당 Promise 객체의 then 메소드로 콜백을 등록하는 새로운 방식**

- promise 객체
  - response.text(), json() ⇒ 모두 promise 객체 리턴
  - promise.then() 형식임
  - 작업의 상태를 가짐

- pending 진행중
- fulfilled 작업 성공 ⇒ response(첫번째 콜백의 파라미터)로 작업 성공 결과를 가짐
- rejected 실패 ⇒ 작업 실패 정보를 가짐
  - req을 보내면 바로 then으로 fulfilled가 될 때! 수행할 콜백 등록
  - resp가 오면 pending → fulfilled 이 되고 콜백 실행
- promise chaining
  - then을 연속적으로 붙이는 것
    - 이전 콜백 리턴값을 다음 콜백 파라미터로 받음
  - then은 각각 다른 promise 객체를 리턴
    - 콜백에서, promise 객체 리턴하는 경우 ⇒ (response.text())
    - 콜백 내 리턴하는 객체의 상태에 따라 fulfilled가 될지 여부와 결과를 그대로 따라서 가짐
    - 콜백에서, 숫자 문자열 일반객체를 리턴하는 경우
      - 무조건 fulfilled, 결과는 그 리턴 값
  - 왜 사용?
    - 비동기 작업을 순차적으로 진행하기 위해서 =
    - 첫번째 작업이 수행되고 나서야 두번째 작업을 실행해야 하는 비동기 작업인 경우 then을 계속 붙여 깔끔하게

```

36 fetch('https://jsonplaceholder.typicode.com/users')
37 .then((response) => response.text())
38 .then((result) => {
39 const users = JSON.parse(result);
40 const { id } = users[0];
41 return fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`);
42 })
43 .then((response) => response.text())
44 .then((posts) => {
45 console.log(posts);
46 });

```

- 예제

```

fetch('https://learn.codeit.kr/api/interviews/summer')
 .then((response) => response.json()) //비직렬화 안하고 바로 사용 가능
 .then((interviewResult) => {
 const { interviewees } = interviewResult;
 const newMembers = interviewees.filter((interviewee) => interviewee.result === 'pass');
 return newMembers; //객체 (배열임)
}) //여기에서 합격한 멤버 선별하여 GET 해서 배열로 리턴
 .then((newMembers) => fetch('https://learn.codeit.kr/api/members', {
 method: 'POST',
 body: JSON.stringify(newMembers)
})) // json 형식으로 바꿔서 멤버사이트에 추가 함
 .then((response) => { // !! post 한 결과의 response는 undefined
 if (response.status === 200) { //정상적으로 추가됐다
 return fetch('https://learn.codeit.kr/api/members') //전체 멤버 조회
 } else {
 throw new Error('New members not added');
 }
}) //잘 추가 된거 확인하고 최신화된 회사 전멤버 url fetch
 .then((response) => response.json())
 .then((members) => {
 console.log(`총 직원 수: ${members.length}`);
 console.log(members);
}); //총 직원 수 + 새로운 멤버들 출력

```

- rejected 상태가 되면

then의 2번째 파라미터로 콜백 적어줌 = 파라미터로 실패 이유 넘어옴

- 예시로 wifi를 끄거나 없는 url 적으면 에러

```
fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => response.json(), (error) => 'Try again!')
 .then((result) => { console.log(result) });
```

- catch

then의 2번째 파라미터에 넣는 방법 말고도 rejected 되었을 때 실행할 메소드

```
fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => response.text())
 .catch((error) => { console.log(error); })
//.then(undefined, (error) => { console.log(error); }) 과 동일
 .then((result) => { console.log(result); }); //위 reject면 undef 찍힘
```

- catch 메소드를 맨 마지막에 쓰면 위 모든 에러를 대응할 수 있음

```
fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => response.json())
 .then((result) => {
 console.log(result);
 // throw new Error('too long');
 })
 .then((result) => {
 console.log(result);
 // throw new Error('no required field');
 })
 .catch((error) => {
 console.log(`[${error.name}]: ${error.message}`);
 });
});
```

▼ 위 에러에 대한 해설

**catch** 메소드 이전에 존재하는 4개의 Promise 객체를 순서대로 하나씩 rejected 상태로 만들어봅시다. 하나의 Promise 객체를 rejected 상태로 만들기 전에 다른 Promise 객체는 fulfilled 상태가 되게끔 다시 수정해 줘야 합니다. 이 점에 주의하고 아래 내용을 읽어 보세요.

### 1. 첫 번째 Promise 객체 rejected 상태로 만들기

```
fetch('https://jsonplaceholder.typicode.commmmm/users')
...
 .catch((error) => {
 console.log(`[${error.name}]: ${error.message}`);
 });
});
```

존재하지 않는 URL을 `fetch()` 함수의 파라미터에 전달하면, `fetch()` 함수의 작업이 실패하고 `fetch()` 함수가 리턴한 Promise 객체의 상태도 rejected 상태가 될 것입니다. 이렇게 고치고 코드를 실행해보면, 아래와 같은 에러 메시지가 출력됩니다.

```
FetchError: request to https://jsonplaceholder.typicode.commmmm/users failed, reason: getaddrinfo ENOTFOUND jsonplaceholder.typicode.commmmm
```

URL 주소에 문제가 있어서 리퀘스트를 보낼 수 없다는 내용이죠?

### 2. 두 번째 Promise 객체 rejected 상태로 만들기

```
fetch('https://google.com')
 .then((response) => response.json())
 ...
 .catch((error) => {
 console.log(`[${error.name}]: ${error.message}`);
 });
});
```

`response.json()`에서 에러가 나게 하려면 어떻게 해야 할까요? JSON 말고 다른 타입의 데이터를 리스폰스의 바디에 담아서 보내주는 URL로 리퀘스트를 보내면 되겠죠? `fetch()` 함수의 URL에 HTML 코드 등을 리스폰스로 주는 구글 홈페이지 URL을 입력하고 실행해 보겠습니다.

```
FetchError: invalid json response body at https://www.google.com/ reason: Unexpected token < in JSON at position 0
```

실행해 보면 위와 같은 에러 정보가 출력됩니다. '유효하지 않은 JSON 리스폰스 바디'라는 에러 메시지를 볼 수 있네요.

### 3. 세 번째 Promise 객체 rejected 상태로 만들기

```
...
 .then((result) => {
 console.log(result);
 throw new Error('too long');
 })
...
 .catch((error) => {
 console.log(`${error.name}: ${error.message}`);
 });
}
```

여기서부터는 인위적으로 에러 객체를 `throw`하는 부분인데요.

에러 객체를 `throw`하는 기준 코드의 주석을 해제해 주면 됩니다. 그리고 코드를 실행하면 앞 부분에 리스폰스의 내용이 잘 출력되고, 그 뒤에 에러 정보가 출력됩니다.

```
[리스폰스의 내용]
Error: too long
```

### 4. 네 번째 Promise 객체 rejected 상태로 만들기

```
...
 .then((result) => {
 throw new Error('no required field');
 })
 .catch((error) => {
 console.log(`${error.name}: ${error.message}`);
 });
}
```

3번과 마찬가지로 에러 객체를 `throw`하는 부분을 주석 해제하고 실행해 보면 앞 부분에 리스폰스의 내용이 잘 출력되고, 그 뒤에 에러 정보가 출력됩니다.

```
[리스폰스의 내용]
Error: no required field
```

방금 한 것처럼 `catch` 메소드에서는 콜백으로 전달된 **Error** 객체를 조사함으로써 Promise Chain 중 어디에서 문제가 발생했는지 판단할 수 있습니다.

```
...
 .catch((error) => {
 if(error.message === 'A'){
 ...
 }else if(error.message === 'B'){
 ...
 }else if(error.message === 'C'){
 ...
 }else{
 ...
 }
 });
}
```

만약 필요하다면 예를 들어, 위와 같은 식으로 각 에러마다 적합한 작업(로깅, 회복 작업 등)을 나누어서 처리해줄 수도 있겠죠? (실전에서는 좀더 깔끔한 방식으로 저 부분을 함수로 따로 만들든지 할 겁니다.)

그리고 이번 실습에서는 에러 객체를 만들기 위해 단순히 `new Error`를 사용했죠.

```

...
 .catch((error) => {
 if(error instanceof TypeError){
 ...
 } else if(error instanceof CustomErrorType_A){
 ...
 } else if(error instanceof CustomErrorType_B){
 ...
 } else{
 ...
 }
 });

```

하지만 여러분이 자바스크립트로 나만의 에러 객체(Custom Error)를 만드는 방법을 알게 되면, 위의 코드처럼 파라미터로 넘어온 에러 객체에 **instanceof**라는 연산자를 붙여서 어느 타입의 에러 객체인지를 좀더 세련된 방식으로 확인할 수 있습니다. 이 내용은 Custom Error 객체를 만드는 방법에 대해 공부하고 나서 다시 살펴보세요.

- 콜백이 리턴하는 값에 따라 **then** 메소드가 리턴한 Promise 객체가 어떻게 달라지는지?
  - 실행된 콜백이 어떤 값을 리턴하는 경우
    - promise 객체 리턴
 

콜백이 리턴한 Promise 객체의 상태와 결과를 똑같이 따라 가짐
    - 이외 숫자, 일반 객체 등 리턴
 

Promise 객체는 fulfilled 상태가 되고 작업 성공 결과로 그 값을 가짐
  - —아무 값도 리턴 x 경우
 

Promise 객체는 fulfilled 상태가 되고, 그 작업 성공 결과로 undefined 를 가짐
  - 실행된 콜백 내부 에러 발생 경우
 

Promise 객체는 rejected 상태가 되고, 그 작업 실패 정보로 해당 Error 객체를 가짐
  - —아무런 콜백이 실행되지 않음
 

이전 Promise 객체와 동일한 상태와 결과를 가짐

```

fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => {
 // return response.json(); // <- Case(1) promise 리턴
 // return 10; // <- Case(2) promise 아닌걸 리턴
 // <- Case(3) 콜백에서 아무것도 리턴 안함
 // throw new Error('failed'); // <- Case(4) 콜백 내에서 에러
 })
 .then((result) => {
 console.log(result);
 });

// 존재하지 않는 URL
fetch('https://jsonplaceholder.typicode.commmmmmm/users')
 .then((response) => response.json())
 // <- Case(5) 콜백 실행 안됨 (원래 2번째 파라미터에 콜백 있어야 하는데, 그러면 아랫줄은 이전, 곧 fetch가 리턴한 promise 객체처럼 reject와 그 결
 .then((result) => { }, (error) => { console.log(error) });

```

#### ▼ 위 5가지 경우

**then** 메소드가 리턴한 Promise 객체를 A라고 했을 때, 각 경우에 A는 다음과 같은 상태와 결과를 갖게 됩니다.

##### Case(1) : 콜백에서 Promise 객체를 리턴

콜백이 리턴한 Promise 객체를 B라고 하면 A는 B와 동일한 상태와 결과를 갖게 됩니다. 나중에 B가 fulfilled 상태가 되면 A도 똑같이 fulfilled 상태가 되고 동일한 작업 성공 결과를, 나중에 B가 rejected 상태가 되면 A도 똑같이 rejected 상태가 되고 동일한 작업 실패 정보를 가진다는 뜻입니다.

##### Case(2) : 콜백에서 Promise 객체가 아닌 일반적인 값을 리턴

A는 fulfilled 상태가 되고, 해당 리턴값을 작업 성공 결과로 갖게 됩니다.

##### Case(3) : 콜백에서 아무것도 리턴하지 않음

자바스크립트에서는 함수가 아무것도 리턴하지 않으면 undefined를 리턴한 것으로 간주합니다.

따라서 A는 fulfilled 상태가 되고, undefined를 작업 성공 결과로 갖게 됩니다.

#### Case(4) : 콜백 실행 중 에러 발생

A는 rejected 상태가 되고, 해당 에러 객체를 작업 실패 정보로 갖게 됩니다.

#### Case(5) : 콜백이 실행되지 않음

A는 호출된 `then` 메소드의 주인에 해당하는, 이전 Promise 객체와 동일한 상태와 결과를 가집니다.

Promise 객체 공부는 `then` 메소드가 그 처음과 끝이라고 해도 될 정도로, `then` 메소드를 정확하게 이해하는 것이 중요합니다. 지금 각각의 케이스를 잘 기억해 두면, 앞으로의 내용을 훨씬 더 쉽게 이해할 수 있을 겁니다.

- finally

- `fulfi`, `reje` 상관없이 무조건 실행되는 코드
- 파라미터 필요 x
- 보통 `catch` 메소드 바로 뒤에 씀. 맨 마지막이 될 듯

```
let isLoading = true; //
/* ..다른 코드들 */

// const url = 'https://jsonplaceholder.typicode.com/users'; //fulfilled 될 url
const url = 'https://www.google.com'; //rej 될 url

fetch(url)
.then((response) => {
 const contentType = response.headers.get('content-type');
 if (contentType.includes('application/json')) {
 return response.json();
 }
 throw new Error('response is not json data');
}) //json 포함 시 리턴, 아님 에러
.then((result) => {
 console.log(result);
})
.catch((error) => {
 console.log(error);
})
.finally(()=>{ //이부분
 isLoading = false;
 console.log(isLoading) //가장 마지막 위 모든게 끝났으니 false
})

console.log(isLoading) //가장 먼저 실행되네, true임, 위 fetch가 진행되는 중임
/* ..다른 코드들 */
```

- then catch finally 최종 퀴즈

```
fetch('https://www.error.www') // 1
.then((response) => response.text()) // 2
.then((result) => { console.log(result); }) // 3
.catch((error) => { console.log('Hello'); throw new Error 'test'; }) // 4
.then((result) => { console.log(result); }) // 5
.then(undefined, (error) => { }) // 6
.catch((error) => { console.log('JS'); }) // 7
.then((result) => { console.log(result); }) // 8
.finally(() => { console.log('final'); }); // 9
```

1. 우선 처음부터 존재하지 않는 url을 `fetch`했기 때문에 1번 라인에서 리턴되는 Promise 객체는 `rejected`상태가 되고, 2번, 3번 라인에서는 `rejected` 상태인 결과값을 처리해줄 콜백함수가 없기 때문에 그대로 4번 라인까지 내려옵니다.
2. 여기서 `catch` 메소드를 만나 "Hello"를 출력하고, 에러를 고의로 발생시켰기 때문에 또 다시 `rejected` 상태가 된 후 에러 객체를 리턴합니다. 그리고 전과 같은 이유로 이 `rejected` 상태를 처리해주기 위해 6번 라인까지 내려오게 됩니다.
3. 첫번째 인자로 `undefined`를 가지는 `then` 메소드는 `catch` 메소드와 같으니, 두번째 인자에 있는 `(error) => {}` 함수가 실행됩니다. 그러나 아무것도 출력되는게 없고, 역시 아무것도 리턴되지 않습니다. 따라서 자동으로 `fulfilled` 상태에, `undefined`를 리턴값으로 가지게 됩니다.
4. `fulfilled` 상태이므로, 7번 라인은 패스하고 8번 라인에서 `result`가 `undefined`를 받아 그대로 `undefined`가 출력됩니다.
5. 마지막으로 `finally` 메소드에 있는 `final`이 출력됩니다.

(throw new Error 'test') 는 에러이름을 test로 정해준 것 뿐인듯

- Promise 객체 등장 이유 (ES6 2015에 등장)
  - 다른 비동기 함수처럼 `fetch('https://first.com', callback)` 이런 식으로 안하는 이유
  - 여러 비동기 작업 순차적 수행해야 할 때 함수에 콜백을 직접 넣으면

```
fetch('https://first.com', (response) => {
 // Do Something
 fetch('https://second.com', (response) => {
 // Do Something ...
 })
})
```

이런 식으로 콜백 헬, 피라미드 형식, 가독성 떨어짐

- pending, fulfilled, rejected 상태, 작업 성공 결과 및 작업 실패 정보(이유), then, catch, finally 메소드 등과 같은 비동기 작업에 관한 보다 정교한 설계

## 심화

- promisify
  - 직접 promise 객체 만드는 것이 있다
- 처음부터 바로 fulfilled 상태이거나 rejected 상태인 Promise 객체를 만드는 것도 가능

```
const p = new Promise((resolve, reject) => {}); //new와 executor 함수 이용
const p = Promise.resolve('success'); //resolve 메소드
const p = Promise.reject(new Error('fail'));
```

- 예제

```
function pick(menus) { //아래 getRandomMenu에서 쓰이는 함수
 console.log('Pick random menu!');
 const p = new Promise((resolve, reject) => { //두 파라미터
 if (menus.length === 0) { //다른 이유가 아닌 메뉴가 없을 때 reject를 수동적으로 시키겠다
 reject(new Error('Need Candidates'));
 } else {
 setTimeout(() => {
 const randomIdx = Math.floor(Math.random() * menus.length);
 const selectedMenu = menus[randomIdx];
 // resolve Promise 객체를 직접 만들 때, 생성된 Promise 객체를 fulfilled 상태로 만들어, 파라미터로 성공 결과를 전달
 resolve(selectedMenu)
 }, 1000); // 시간이 걸리는 걸 시뮬레이션하기 위한 1초입니다
 }
 });
 return p; //프로미스 객체
}

function getRandomMenu() {
 return fetch('https://learn.codeit.kr/api/menus')
 .then((response) => response.json())
 .then((result) => {
 const menus = result;
 return pick(menus); // ! random pick function
 //프로미스 객체 리턴
 });
}

getRandomMenu() //위에서 프로미스 객체 리턴
 .then((menu) => {
 console.log(`Today's lunch is ${menu.name} ~`);
 })
 .catch((error) => {
 console.log(error.message);
 })
 .finally(() => {
 console.log('Random Menu candidates change everyday');
 });
}
```

- 여러 개의 Promise 객체를 다뤄야 할 때 사용되는 Promise의 메소드

어떤 메소드든 결국 하나의 Promise 객체를 리턴

- all = 여러 Promise 객체의 작업 성공 결과를 기다렸다가 모두 한 번에 취합 하기 위해서 사용, 그 배열 리턴
- race = race가 리턴한 객체는 가장 먼저 fulfilled 상태 또는 rejected 상태가 된 Promise 객체와 동일한 상태와 결과를 가짐
- allSettled = pending 상태의 Promise 객체가 하나도 없게 되면, A의 상태값은 fulfilled 상태가 되고 그 작업 성공 결과로, 하나의 배열을 가짐
- any = 여러 Promise 객체들 중에서 가장 먼저 fulfilled 상태가 된 Promise 객체의 상태와 결과가 A에도 똑같이 반영

- axios

- ajax 통신 가능하게 하는 fetch와 비슷한 패키지
- 장점
  - 모든 리퀘스트, 리스폰스에 대한 공통 설정 및 공통된 전처리 함수 삽입 가능
  - serialization, deserialization을 자동으로 수행
  - 특정 리퀘스트에 대해 얼마나 오랫동안 리스폰스가 오지 않으면 리퀘스트를 취소할지 설정 가능(request timeout)
  - 업로드 시 진행 상태 정보를 얻을 수 있음
  - 리퀘스트 취소 기능 지원
- 단점
  - 별도 다운로드 필요

```
axios
.get('https://jsonplaceholder.typicode.com/users')
.then((response) => {
 console.log(response);
})
.catch((error) => {
 console.log(error);
});
```

---

- async/await

promise chaining을 더 편하게 작성하고, 가독성을 높이기 위한 것

- async
  - 비동기 의미 - 함수 안에 비동기로 실행될 부분이 있다는 걸 의미
- await
  - = 기다리다 - promise 리턴하는 코드 앞에 붙음, promise 객체가 ful or rej 될때까지 기다림, ful이 되면 그 결과인 객체 추출해서 리턴함, 그래야 다음 코드로 넘어감
  - async 함수 안에서만 사용 가능
  - await을 만나는 순간 함수 바깥으로 나가서 함수 호출 이후 코드들을 실행하고 다시 돌아옴

```
async function fetchAndPrint() {
 console.log(2);
 const response = await fetch('https://jsonplaceholder.typicode.com/users');
 console.log(7);
 const result = await response.text();
 console.log(result);
}

console.log(1);
fetchAndPrint();
console.log(3);
console.log(4);
console.log(5);
console.log(6);
----- 동일

function fetchAndPrint() {
```

```

console.log(2);
fetch('https://jsonplaceholder.typicode.com/users')
 .then((response) => {
 console.log(7);
 return response.text();
 })
 .then((result) => { console.log(result); });
}

console.log(1);
fetchAndPrint();
console.log(3);
console.log(4);
console.log(5);
console.log(6);

```

- chain → async 전환 예제

```

fetch("https://jsonplaceholder.typicode.com/users")
 .then((response) => response.json())
 .then((users) => {
 const lastUser = users[users.length - 1];
 return lastUser.id;
 })
 .then((id) => fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`))
 .then((response) => response.json())
 .then((posts) => {
 const lastPost = posts[posts.length - 1];
 console.log(lastPost);
 });

//위 chain을 async/await로 전환한 코드
//await가 있는 줄은 현 파라미터가 이전 then에게 promise 객체 받았을 때임
//const 현재 1파라미터 = await 이전에서 주는 promise 객체
async function getTheLastPostOfTheLastUser() {
 const response = await fetch("https://jsonplaceholder.typicode.com/users")
 const users = await response.json()
 const lastusers = users[users.length - 1];
 const id = lastusers.id; //이것도 넘겨 받은거지만 id는 pr 객체 아님
 const response2 = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${id}`)
 const posts = await response2.json()
 const lastPost = posts[posts.length - 1];
 return lastPost //promise 객체 리턴
}

getTheLastPostOfTheLastUser()
 .then((lastPost) => { console.log(lastPost); });

```

- async 순서 예제

```

const p1 = fetch('https://jsonplaceholder.typicode.com/users?id=1')
 .then((response) => response.text()); //fetch 함수의 리퀘스트에 대한 리스폰스가 있을 때
const p2 = new Promise((resolve, reject) => {
 setTimeout(() => { resolve('Hello'); }, 2000);
}); //2초(2000밀리세컨즈)가 지났을 때
const p3 = Promise.resolve('Success'); //즉시
// const p4 = Promise.reject(new Error('Fail'));

async function test() {
 console.log(await p1);
 console.log(await p2);
 console.log(await p3);
 // console.log(await p4);
}

console.log('----Start----');
test();
console.log('----End----');

//start, end가 먼저 나오고 그 뒤 async func 내에선 누가 먼저 fulfilled 된거 상관없이 순서대로

```

- rej가 되었을때

- try catch 사용
- 함수문 전체 try

- async 함수들 간의 순서 퀴즈

```

async function test1() {
 const result = await Promise.resolve('success');
 console.log(result);
}

async function test2() {
 try {
 const p = new Promise((resolve, reject) => {
 setTimeout(() => { resolve('last'); }, 3000);
 }); //이때 한번 나가고
 const result = await p; //이때 또 나가는건가
 console.log(result);
 } catch (e) {
 console.log(e);
 }
}

async function test3() {
 try {
 const result = await Promise.reject('fail');
 console.log(result);
 } catch (e) {
 console.log(e);
 }
}

test1();
console.log('JavaScript');
test2();
console.log('Programming');
test3();

//JavaScript - Programming - success - fail - last

```

- prompt로 퀴즈 내는 예제

```

async function showQuiz() {
 try {
 const response = await fetch('https://learn.codeit.kr/api/quiz');
 const test = await response.json(); //test가 위 url json 객체임
 const yourAnswer = prompt(test.quiz);
 if (yourAnswer.toLowerCase() === test.answer) {
 alert(`Good Job, ${test.explanation} => Let's learn more with Codeit!`);
 } else {
 throw new Error('wrong');
 }
 } //여기까지가 퀴즈 내주고 정답 or 오답 알려주는 실행코드
 catch (error) {
 if (error.message === 'wrong') { //답 틀리면 주는 에러
 alert('You need to learn JavaScript with Codeit!');
 } else {
 alert('Error');
 }
 }
 finally {
 window.open('https://codeit.kr', '_blank');
 }
}

showQuiz();

```

- async 함수는 무조건 promise 객체를 리턴

▼ async 함수의 promise 객체 리턴값

## 1. 어떤 값을 리턴하는 경우

### (1) Promise 객체를 리턴하는 경우

async 함수 안에서 Promise 객체를 리턴하는 경우에는 해당 Promise 객체와 동일한 상태와 작업 성공 결과(또는 작업 실패 정보)를 가진 Promise 객체를 리턴합니다.(그냥 해당 Promise 객체를 리턴한다고 봐도 괜찮습니다.)

```
async function fetchAndPrint() {
 return new Promise((resolve, reject)=> {
 setTimeout(() => { resolve('abc'); }, 4000);
 });
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 return new Promise((resolve, reject) => {
 setTimeout(() => { resolve('abc'); }, 4000);
 });
}

fetchAndPrint();
▼Promise {<pending>} ⓘ
▶__proto__: Promise
 [[PromiseState]]: "pending"
 [[PromiseResult]]: undefined
```

이렇게 pending 상태의 Promise 객체를 리턴하기도 하고(리턴된 Promise 객체는 약 4초 후에 fulfilled 상태가 되겠죠?)

```
async function fetchAndPrint() {
 return Promise.resolve('Success');
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 return Promise.resolve('Success');
}

fetchAndPrint();
▼Promise {<fulfilled>: "Success"} ⓘ
▶__proto__: Promise
 [[PromiseState]]: "fulfilled"
 [[PromiseResult]]: "Success"
```

이미 fulfilled 상태인 Promise 객체나

```
async function fetchAndPrint() {
 return Promise.reject(new Error('Fail'));
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 return Promise.reject(new Error('Fail'));
}

fetchAndPrint();
Promise {<rejected>: Error: Fail
 ▼ at fetchAndPrint (<anonymous>:2:25)
 at <anonymous>:5:1} ⓘ
 ► __proto__: Promise
 [[PromiseState]]: "rejected"
 [[PromiseResult]]: Error: Fail at fetchAndPrint (<anonymous>:2:25) at <anonymous>:5:1
 ► Uncaught (in promise) Error: Fail
 at fetchAndPrint (<anonymous>:2:25)
 at <anonymous>:5:1
```

이미 rejected 상태인 Promise 객체를 리턴하는 경우 전부 다 해당합니다. (위 이미지에서는 rejected 상태의 Promise 객체를 따로 처리해주지 않았기 때문에 에러가 발생한 겁니다)

## (2) Promise 객체 이외의 값을 리턴하는 경우

async 함수 내부에서 Promise 객체 이외에 숫자나 문자열, 일반 객체 등을 리턴하는 경우에는, **fulfilled** 상태이면서, 리턴된 값을 작업 성공 결과로 가진 Promise 객체를 리턴합니다.

```
async function fetchAndPrint() {
 return 3;
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 return 3;
}

fetchAndPrint();
Promise {<fulfilled>: 3} ⓘ
 ► __proto__: Promise
 [[PromiseState]]: "fulfilled"
 [[PromiseResult]]: 3
```

이런 코드나

```
async function fetchAndPrint() {
 return 'Hello';
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 return 'Hello';
}

fetchAndPrint();
▼ Promise {<fulfilled>: "Hello"} ⓘ
▶ __proto__: Promise
 [[PromiseState]]: "fulfilled"
 [[PromiseResult]]: "Hello"
```

이런 코드,

```
async function fetchAndPrint() {
 const member = {
 name: 'Jerry',
 email: 'jerry@codeitmall.kr',
 department: 'sales',
 };

 return member;
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 const member = {
 name: 'Jerry',
 email: 'jerry@codeitmall.kr',
 department: 'sales',
 };

 return member;
}

fetchAndPrint();
▼ Promise {<fulfilled>: {...}} ⓘ
▶ __proto__: Promise
 [[PromiseState]]: "fulfilled"
 ▼ [[PromiseResult]]: Object
 department: "sales"
 email: "jerry@codeitmall.kr"
 name: "Jerry"
 ▶ __proto__: Object
```

이런 코드를 모두 여기에 해당합니다.

## 2. 아무 값도 리턴하지 않는 경우

```
async function fetchAndPrint() {
 console.log('Hello Programming!');
}

fetchAndPrint();
```

이렇게 함수에서 아무런 값도 리턴하지 않으면 자바스크립트에서 어떻게 간주한다고 했죠? undefined를 리턴한 것으로 간주한다고 했는데요. 따라서

```
async function fetchAndPrint() {
 console.log('Hello Programming!');
}

fetchAndPrint();
Hello Programming!
▼ Promise {<fulfilled>: undefined} ⓘ
 ► __proto__: Promise
 [[PromiseState]]: "fulfilled"
 [[PromiseResult]]: undefined
```

이 경우에는 fulfilled 상태이면서, undefined를 작업 성공 결과로 가진 Promise 객체가 리턴됩니다.

### 3. async 함수 내부에서 에러가 발생했을 때

```
async function fetchAndPrint() {
 throw new Error('Fail');
}

fetchAndPrint();
```

```
async function fetchAndPrint() {
 throw new Error('Fail');
}

fetchAndPrint();
Promise {<rejected>: Error: Fail
 ▼ at fetchAndPrint (<anonymous>:2:9)
 at <anonymous>:5:1} ⓘ
 ► __proto__: Promise
 [[PromiseState]]: "rejected"
 [[PromiseResult]]: Error: Fail at fetchAndPrint (<anonymous>:2:9) at <anonymous>:5:1
▶ Uncaught (in promise) Error: Fail
 at fetchAndPrint (<anonymous>:2:9)
 at <anonymous>:5:1
```

async 함수 안에서 에러가 발생하면, rejected 상태이면서, 해당 에러 객체를 작업 실패 정보로 가진 Promise 객체가 리턴됩니다.

자, 이때까지 async 함수 안에서 리턴하는 값에 따라, async 함수가 결국 어떤 Promise 객체를 리턴하는지 배웠는데요. 이전에 '[then 메소드 완벽하게 이해하기](#)' 노트에서 배운 내용과 비슷해서 별로 어렵지 않죠?

이렇게 async 함수가 결국 Promise 객체를 리턴한다는 사실은 아주 중요합니다. 왜냐하면 이 말은 곧 async 함수 안에서 다른 async 함수를 가져다가 쓸 수 있다는 뜻이기 때문입니다. 이 말이 무슨 뜻인지 다음 영상에서 알아봅시다.

- fetch then과 거의 비슷
- async 안에 async
  - 민감 정보 조회 제외
  - async 안에 async 가 가능한 것은, async는 promise 리턴하므로 그 앞에 await 붙여 쓰면됨
- async 붙이는 위치

```

// 1) Function Declaration
async function example1(a, b) {
 return a + b;
}

// 2-1) Function Expression(Named)
const example2_1 = async function add(a, b) {
 return a + b;
};

// 2-2) Function Expression(Anonymous)
const example2_2 = async function(a, b) {
 return a + b;
};

// 3-1) Arrow Function
const example3_1 = async (a, b) => {
 return a + b;
};

// 3-2) Arrow Function(shortened)
const example3_2 = async (a, b) => a + b;

```

- 점심 메뉴 랜덤 생성기 예제

```

async function pick(menus) { //
 console.log('Pick random menu!');
 const p = new Promise((resolve, reject) => {
 if (menus.length === 0) {
 reject(new Error('Need Candidates'));
 } else {
 setTimeout(() => {
 const randomIdx = Math.floor(Math.random() * menus.length);
 const selectedMenu = menus[randomIdx];
 resolve(selectedMenu);
 }, 1000);
 }
 });
 return p; //Promise 객체 리턴하는 함수므로 async 붙여줌
}

async function getRandomMenu() {
 console.log('---Please wait!---'); //에러 날만한 것만 try 안에
 try {
 const response = await fetch('https://learn.codeit.kr/api/menus')
 const result = await response.text()
 const menus = JSON.parse(result);
 const menu = await pick(menus); //여기서 위 async 함수 부름
 console.log(`Today's lunch is ${menu.name}`);
 } catch (e) {
 console.log(e.message);
 } finally {
 console.log('Random Menu candidates change everyday');
 }
} //try catch finally로 아래의 chaining 대체 (한꺼번에 써주는 것으로)

getRandomMenu() //함수 call 하기만 하면 됨
////////////////////

function pick(menus) {
 console.log('Pick random menu!');
 const p = new Promise((resolve, reject) => {
 if (menus.length === 0) {
 reject(new Error('Need Candidates'));
 } else {
 setTimeout(() => {
 const randomIdx = Math.floor(Math.random() * menus.length);
 const selectedMenu = menus[randomIdx];
 resolve(selectedMenu);
 }, 1000);
 }
 });
 return p;
}

function getRandomMenu() {
 console.log('---Please wait!---');
 return fetch('https://learn.codeit.kr/api/menus')
 .then((response) => response.text())
 .then((result) => {

```

```

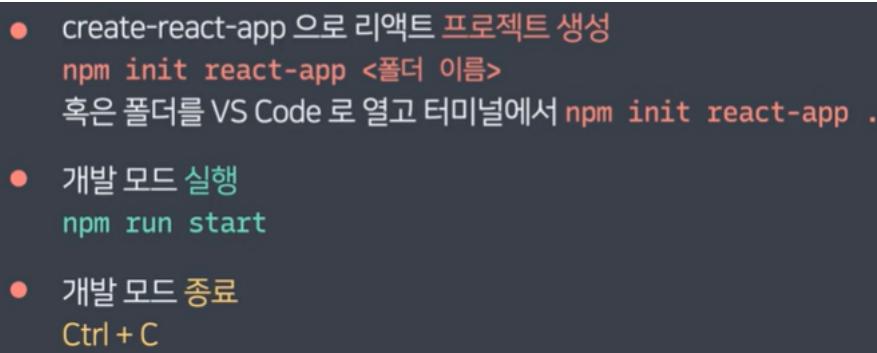
 const menus = JSON.parse(result);
 return pick(menus); // ! random pick function
 });

getRandomMenu()
 .then((menu) => {
 console.log(`Today's lunch is ${menu.name}`);
 })
 .catch((error) => {
 console.log(error.message);
 })
 .finally(() => {
 console.log('Random Menu candidates change everyday');
 });

```

## ▼ 리액트 웹 개발

- nodejs = 브라우저 밖에서도 js 실행시킬 수 있게, fe는 자주 쓰이는 기능만 알아두자
- npm - node 패키지 매니저 - 노드에서 실행할 패키지 관리하고 실행하는 도구



- 리액트 18버전 이후 `ReactDOM.render` 가 아니라 `ReactDOM.createRoot`

- 프로젝트 시 폴더 하나 만들고 init 해줘야하는듯

- public에서 index.html 제외 다 삭제

- index도 다음과 같이 정리

```

<!DOCTYPE html>
<html lang="ko">
 <head>
 <meta charset="utf-8" />
 <title>주사위 게임</title>
 </head>
 <body>
 <div id="root"></div>
 </body>
</html>

```

- src에서도 index.js 제외 삭제

- index.js도 정리

```

import React from 'react';
import ReactDOM from 'react-dom/client';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
 <h1>⚇ ⚈ ⚉ ⚊</h1>
);

----- 이것과 동일
ReactDOM.render(
 <h1>⚇ ⚈ ⚉ ⚊</h1>
, document.getElementById('root'));

```

- index.html이 가장 먼저 실행됨, 그 이후 js가 실행됨 (리액트 중 가장 먼저)

- jsx - js 안에 html 문법 넣는 새로운 문법

- class - 객체 지향의 것이라  
className 로 접근해야함
- for - label에서 사용 시 htmlFor로
- jsx 프로퍼티는 항상 두번째 단어는 대문자로 = camelCase로
- 꼭 하나의 태그 div 등으로 감싸야함
  - 안그라고 싶다면? <Fragment>로 감싸기 - 개발자 도구에선 안보임
  - <> </> 이렇게 빈 태그로 감싸도됨

```
import ReactDOM from 'react-dom';

ReactDOM.render(
 <> // 웬만하면 빈 태그 활용
 <h1 id="title">가위바위보</h1>
 <button className="hand">가위</button> // className
 <button className="hand">바위</button>
 <button className="hand">보</button>
 </>
, document.getElementById('root'));
```

- 실행 시에 jsx는 js로 변환되어 실행됨, 따라서 내부에서 js도 사용 가능 (중괄호 내에는 표현식만 가능)
  - <h1 id="title"> \${product.toUpperCase()} </h1>
  - <img src={ 위에서 선언한 js 변수 url }>
  - <button onClick={ 위에서 선언한 함수이름 }>

```
import ReactDOM from 'react-dom';

const WINS = {
 rock: 'scissor',
 scissor: 'paper',
 paper: 'rock',
};

function getResult(left, right) {
 if (WINS[left] === right) return '승리';
 else if (left === WINS[right]) return '패배';
 return '무승부';
}

function handleClick() {
 console.log('가위바위보!');
}

const me = 'rock';
const other = 'scissor';

ReactDOM.render(
 <>
 <h1>가위바위보</h1>
 <h2>{getResult(me, other)}</h2> // 리턴값
 <button onClick={handleClick}>가위</button> // 이벤트 함수
 <button onClick={handleClick}>바위</button>
 <button onClick={handleClick}>보</button>
 </>,
 document.getElementById('root')
);
```

- 컴포넌트

- function 형태의 첫글자 대문자, jsx를 리턴해야
- 세분화, 재사용의 장점

```

import diceBlue01 from './assets/dice-blue-1.svg';
//asset에서 이미지 import

function Dice() {
 return ;
 //diceBlue01 변수명 안쓰고 주소 그대로 쓰면 액박 뜸
}

export default Dice;
-----Dice를 App이 넘겨받음
import Dice from './Dice';

function App() {
 return (
 <div>
 <Dice /> //컴포넌트는 이렇게 써임
 </div>
);
}
//dice.js 컴포넌트 import 해 띄워주고 export

export default App;
-----App을 index.js가 넘겨받음
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
//결국 이게 html을 담는 것

```

- Props

- 같은 컴포넌트를 쓰면서 차이를 둘 수 있따

```

import rockImg from './assets/rock.svg';
import scissorImg from './assets/scissor.svg';
import paperImg from './assets/paper.svg';

function HandIcon({value}) { //props으로 받고 props.value로 접근도 가능
 if(value=='rock')
 return ;
 else if (value=='scissor')
 return ;
 else if (value=='paper')
 return ;
}

export default HandIcon;

//더 좋은 방법
const IMAGES = {
 rock: rockImg,
 scissor: scissorImg,
 paper: paperImg,
};

function HandIcon({ value }) {
 const src = IMAGES[value];
 return ;
}
-----App.js
import HandIcon from './HandIcon';

function App() {
 return (
 <div>
 <HandIcon value="rock" />
 <HandIcon value="scissor" />
 <HandIcon value="paper" />
 </div>
);
}

export default App;

```

- prop으로 아이콘 누르면 해당 아이콘 원지 로그띄우는 예제

```

import HandButton from './HandButton';

function App() {
 const handleClick = (value) => console.log(value);
 return (
 <div> //HandButton으로 prop 2개 내려줌
 <HandButton value="rock" onClick={handleClick} />
 <HandButton value="scissor" onClick={handleClick} />
 <HandButton value="paper" onClick={handleClick} />
 </div>
);
}

export default App;

import HandIcon from './HandIcon';

//다른 컴포넌트의 prop으로 다시 내려 주는 경우, 2번 전달
function HandButton({ value, onClick }) {
 const handleClick = () => onClick(value); //onClick은 여기 이벤트리스너에 써줌
 return (//app에서 받은 2개 prop 중 onClick은 여기 버튼에서 쓰고 value는 HandIcon에 한번더 전달
 <button onClick={handleClick}>
 <HandIcon value={value} />
 </button>
)
}

export default HandButton;

import rockImg from './assets/rock.svg';
import scissorImg from './assets/scissor.svg';
import paperImg from './assets/paper.svg';

const IMAGES = {
 rock: rockImg,
 scissor: scissorImg,
 paper: paperImg,
};

function HandIcon({ value }) { //value 하나 받음 - 이미지 띄워줌
 const src = IMAGES[value];
 return ;
}

export default HandIcon;

```

- children 프로퍼티

```

function Button({text, func1})
{ return <button onClick={func1}>{text}</button> }

function App()
{ return <Button text="버튼1" onClick=~>/> }

```

```

function Button({children, func1})
{ return <button onClick={func1}>{children}</button> }

function App()
{ return <Button onClick=~>버튼1</Button> }

```

- jsx에선 props.children 으로 접근 - innerText가 아닌

- state

- 변수 개념

```

import Button from './Button';
import HandButton from './HandButton';
import HandIcon from './HandIcon';
import { compareHand, generateRandomHand } from './utils'; //다른 모듈에서 함수 끌어다씀
import {useState} from 'react' //useState쓰려면

function getResult(me, other) { //가위바위보 결과 내는 함수
 const comparison = compareHand(me, other);

```

```

 if (comparison > 0) return '승리';
 if (comparison < 0) return '패배';
 return '무승부';
 }

 function App() {
 // hand와 otherHand를 state로 바꿔 주세요 - 내와 상대 상태
 const [hand, setHand] = useState('rock')
 const [otherHand, setOtherHand] = useState('scissor')

 // 누른 버튼의 상태를 파라미터로 받아서
 const handleButtonClick = (nextHand) => {
 // hand의 값을 nextHand로 바꿔 주세요
 setHand(nextHand);
 // otherHand의 값을 generateRandomHand()의 리턴 값으로 바꿔주세요
 setOtherHand(generateRandomHand());
 };

 const handleClearClick = () => { // 둘다 rock으로 초기화
 // hand와 otherHand의 값을 'rock'으로 바꿔주세요
 setHand('rock')
 setOtherHand('rock')
 };

 return (
 <div>
 <Button onClick={handleClearClick}>처음부터</Button>
 <p>{getResult(hand, otherHand)}</p>
 <div>
 <HandIcon value={hand} />
 VS
 <HandIcon value={otherHand} />
 </div>
 <div> // 각 버튼 누르면 이벤트 핸들러로 아래 함수 부름
 <HandButton value="rock" onClick={handleButtonClick} />
 <HandButton value="scissor" onClick={handleButtonClick} />
 <HandButton value="paper" onClick={handleButtonClick} />
 </div>
 </div>
);
 }

 export default App;

```

- 참조형 state

render가 state가 변해야 되는건데

배열에서 gameHistory.push(nextNum) setGameHistory(gameHistory) 하면 배열 요소는 추가되어도 어차피 저 값은 주 소값이라 안바뀜

→ 새로운 값을 만들어서 변경해야

setGameHistory([...gameHistory, nextNum])

```

import { useState } from 'react';
import Button from './Button';
import HandButton from './HandButton';
import HandIcon from './HandIcon';
import { compareHand, generateRandomHand } from './utils';

const INITIAL_VALUE = 'rock';

function getResult(me, other) {
 const comparison = compareHand(me, other);
 if (comparison > 0) return '승리';
 if (comparison < 0) return '패배';
 return '무승부';
}

function App() {
 const [hand, setHand] = useState(INITIAL_VALUE);
 const [otherHand, setOtherHand] = useState(INITIAL_VALUE);
 const [gameHistory, setGameHistory] = useState([]); // 처음 빈배열로 초기화

 // 이제 누른 버튼의 상태를 가지고 나와 상대 상태 설정하고
 // 그 둘의 상태로 결과를 내고 그걸 gameHistory에 기록
 const handleButtonClick = (nextHand) => {
 const nextOtherHand = generateRandomHand();
 const nextHistoryItem = getResult(nextHand, nextOtherHand);
 setHand(nextHand);
 setOtherHand(nextOtherHand);
 // gameHistory에 nextHistoryItem 을 추가해 주세요
 }
}

```

```

 // 이렇게 spread 구문 쓴 이유는 이렇게 배열 매번 새로 만들어줘야 render됨
 setGameHistory([...gameHistory, nextHistoryItem])
 //set배열([...기존배열, 추가할 요소])
};

const handleClearClick = () => {
 setHand(INITIAL_VALUE);
 setOtherHand(INITIAL_VALUE);
 // gameHistory를 비워주세요
 setGameHistory([])
};

return (
 <div>
 <Button onClick={handleClearClick}>처음부터</Button>
 <div>
 <HandIcon value={hand} />
 VS
 <HandIcon value={otherHand} />
 </div>
 <p>승부 기록: {gameHistory.join(', ')})</p> //gameHistory를 표시하되 , 로 구분
 <div>
 <HandButton value="rock" onClick={handleButtonClick} />
 <HandButton value="scissor" onClick={handleButtonClick} />
 <HandButton value="paper" onClick={handleButtonClick} />
 </div>
 </div>
);
}

export default App;

```

```

import { useState } from 'react';
import Button from './Button';
import HandButton from './HandButton';
import HandIcon from './HandIcon';
import { compareHand, generateRandomHand } from './utils';

const INITIAL_VALUE = 'rock';

function getResult(me, other) {
 const comparison = compareHand(me, other);
 if (comparison > 0) return '승리';
 if (comparison < 0) return '패배';
 return '무승부';
}

function App() {
 const [hand, setHand] = useState(INITIAL_VALUE);
 const [otherHand, setOtherHand] = useState(INITIAL_VALUE);
 const [gameHistory, setGameHistory] = useState([]);
 const [score, setScore] = useState(0); //내 스코어
 const [otherScore, setOtherScore] = useState(0); //상대 스코어
 const [bet, setBet] = useState(1); //베팅포인트

 const handleButtonClick = (nextHand) => {
 const nextOtherHand = generateRandomHand();
 const nextHistoryItem = getResult(nextHand, nextOtherHand);
 const comparison = compareHand(nextHand, nextOtherHand);
 setHand(nextHand);
 setOtherHand(nextOtherHand);
 setGameHistory([...gameHistory, nextHistoryItem]);
 if (comparison > 0) setScore(score + bet); //이기면 내 스코어+벳
 if (comparison < 0) setOtherScore(otherScore + bet); //
 };

 const handleClearClick = () => {
 setHand(INITIAL_VALUE);
 setOtherHand(INITIAL_VALUE);
 setGameHistory([]);
 setScore(0);
 setOtherScore(0);
 setBet(1);
 };

 //이벤트 핸들러에서 input 의 value 속성을 참조하려면 e.target.value
 //원래 js에선 input값 변경시 oninput으로 처리하지만 리액트는 onChange로 통합
 const handleBetChange = (e) => {
 // 여기에 코드를 작성하세요, input 값이 달라지면 이값을 bet으로 설정
 setBet(Number(e.target.value))
 };

 //예시 모범답 - 문자열 입력시 value가 이미 빈문자열이라 num이 0됨
 // let num = Number(e.target.value);
}

```

```

// if (num > 9) num %= 10; // 1과 9 사이의 숫자로 만들어 줌
// if (num < 1) num = 1;
// num = Math.floor(num);
// setBet(num);

return (
 <div>
 <Button onClick={handleClearClick}>처음부터</Button>
 <div>
 {score} : {otherScore}
 </div>
 <div>
 <HandIcon value={hand} />
 VS
 <HandIcon value={otherHand} />
 </div>
 <div>
 <input onChange={handleBetChange} type="number" value={bet} min={1} max={9}></input>
 </div>
 <p>승부 기록: {gameHistory.join(', ')</p>
 <div>
 <HandButton value="rock" onClick={handleButtonClick} />
 <HandButton value="scissor" onClick={handleButtonClick} />
 <HandButton value="paper" onClick={handleButtonClick} />
 </div>
 </div>
);
}

export default App;

```

- 컴포넌트 (자동차 부품)
  - 반복적 일이 줄어듬 - 재활용
  - 고장 고치기 쉽다
  - 일을 쉽게 나눔 - 여러 사람이 분담 가능
- F2 키로 한꺼번에 바꾸기
- state lifting
  - 자식 것을 부모로 올리는 것
  - 두 자식을 공통으로 컨트롤 할 수 있는 버튼을 만들때
- 리액트 렌더링 방식
  - return문의 것들을 계속 새로 리턴
  - 전부가 아닌 virtual dom 형식 이용
    - 단순 깔끔 코드 작성 가능
    - 변경사항들 효율적 처리 가능
    - (virtual dom에 먼저 적용하고 dom에 그 다음으로 적용)
- 리액트 jsx에 직접 스타일 입히기

<button style={style1}> 이런 형식으로,  
위에서 style1은 객체 형식으로, camelCase로 선언

```

import HandIcon from './HandIcon';
import img1 from './assets/purple.svg'

// 객체니까 string은 모두 걸 따옴표
// 이미지 주소는 import 해서 변수 안에 넣고 `url(${변수})` 사용
const style={
 width: '166px',
 height: '166px',
 border: 'none',
 outline: 'none',
 textAlign: 'center',
}

```

```

 cursor: 'pointer',
 backgroundColor: 'transparent',
 backgroundImage: `url(${img1})`,
 backgroundRepeat: 'no-repeat',
 backgroundPosition: 'center',
 backgroundSize: 'contain',
 }

// 인라인 스타일을 적용해 주세요
function HandButton({ value, onClick }) {
 const handleClick = () => onClick(value);
 return (
 <button style={style} onClick={handleClick}>
 <HandIcon value={value} />
 </button>
);
}

export default HandButton;

```

- 리액트에 css 파일 적용해서 style 입하기
  - js 파일에서 css 파일 임포트만 하면 바로 적용
    - index.js에 import './index.css' 하고 className으로 접근

```

function HandButton({ value, onClick }) {
 const handleClick = () => onClick(value);
 return (
 <button className="HandButton" onClick={handleClick}>
 <HandIcon className="HandButton-icon" value={value} />
 </button>
);
}

function HandIcon({ className, value }) {
 const src = IMAGES[value];
 return ;
}

```

- 리액트에서 인라인스타일 vs css 파일 적용?
 

:hover(마우스 올린 상태)나 :focus(입력창 포커스된 상태)같은 CSS에서 상태를 처리하는 경우에는 인라인 스타일로 할 수 없어서 꼭 CSS로 작성해야할 거 같아요!

하지만 반대로 인라인 스타일에서만 할 수 있는 것들이 있는데 CSS 파일은 고정된 값으로 들어가지만 인라인 스타일은 자바스크립트라서 변하는 값을 지정할 수 있을 거 같습니다.

어떤 사이트들은 아바타 이미지 올릴 때 보시면 잘라내기 한다던가 할 때 영역을 표시해주는데 이런건 변하는 값이라서 인라인 스타일을 쓰면 되지 않을까요?

둘 다 쓸 수 있는 경우라면 CSS 클래스를 쓰는 게 CSS의 상속 기능이나 등등 다양한 것을 활용할 수 있어서 장점이 될 거 같습니다!

  - 인라인
    - 위에 style 객체로 선언 후 style={style}
  - CSS
    - import '~.css' 후 className={css 내에 있는 class 이름}

---

빌드하기

## 프로젝트 생성하기

터미널에서 원하는 디렉토리에 들어가서 `npm init react-app .` 를 입력하면 현재 디렉토리에 리액트 프로젝트를 생성합니다.

## 개발 모드 실행하기

터미널에서 `npm run start` 를 입력하면 개발 모드 서버가 실행됩니다.

## 실행 중인 서버 종료하기

서버가 실행 중인 터미널에서 `ctrl + c` 를 입력하면 서버가 종료됩니다.

## 개발된 프로젝트 빌드하기

터미널에서 `npm run build` 를 입력하면 빌드를 시작합니다.

## 빌드한 것 로컬에서 실행하기

터미널에서 `npx serve build` 를 입력하면 serve 프로그램을 다운 받고 build 폴더에서 서버가 실행됩니다.

### ▼ 리액트로 데이터 다루기

- 리액트는 렌더링을 자동으로 해주니 편리 - 유튜브 등
- vscode 단축키
  - 클릭, 더블클릭 다름
  - 알트 누르면 수동 중복 커서
  - 컨 쉬 !
  - handleClick - hc로 자동완성
  - f2 이름바꾸기
  - 알트 + 위아래
  - 쉬 탭 (내어쓰기)
  - 세팅에서 디폴트 설정 시 프리티어 - 저장할때마다 코드 정돈됨 ctrl+s

---

### movie-app

- map의 콜백함수에서 jsx로 작성한 것을 리턴하는 방법 - 많은 데이터를 보여줄 때 유용한 방법
  - 배열을 렌더링할 때는 `map` 을 쓰자!

```
function ReviewList ({items}) {
 return (
 <div>

 {items.map((item) => { //map 함수는 두번째로 콜백함수
 return {item.title}
 })}

 </div>
)
}

export default ReviewList;
```

```
import "./ReviewList.css"

function ReviewListItem ({item}) {
 return (
 <div>

 <div>
 <p>{item.id}</p>
 <p>{item.rating}</p>
 <p>{item.content}</p>
 </div>
 </div>
)
}

function ReviewList ({items}) {
 return (
```

```

 <div>

 {items.map((item) => {
 return <ReviewListItem item={item} />
))}

 </div>
)
 }

export default ReviewList;

```

- 정렬 기능

```

import ReviewList from "./ReviewList";
import items from "../mock.json"
import { useState } from "react";

function App() {
 //리뷰리스트.js가 아닌 app에서 하는건 버튼으로 조절 가능하고, 다른 순서로도 정렬 컨트롤 하게 되도록
 const [sort, setSort] = useState("id")
 const sortedItem = items.sort((a,b) => b[sort]-a[sort])

 //각 함수 내에서 sort 함수 쓰니까 안됨 ;;
 //하나의 sortedItem 함수를 공용으로 쓰고 id, rating으로 할지를 정해줌
 const handleRatingSort = () => {
 setSort("rating")
 }

 const handleIdSort = () => {
 setSort("id")
 }

 return (
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>
 <ReviewList items={sortedItem} /> //어떻게든 정렬된 아이템을 받으니
 </div>
);
}

export default App;

```

- 삭제 기능 !! 중요 !!

```

import ReviewList from "./ReviewList";
import mockItems from "../mock.json"
import { useState } from "react";

function App() {
 const [sort, setSort] = useState("id")
 const [items, setItems] = useState(mockItems) //배열 수정도 useState로
 const sortedItem = items.sort((a,b) => b[sort]-a[sort])

 const handleRatingSort = () => {
 setSort("rating")
 }

 const handleIdSort = () => {
 setSort("id")
 }

 //App에서 이걸 하는 이유는 sort와 마찬가지, items 전체에서 item 한 묶음을 삭제해야하니?
 //id를 파라미터로 가겠지만 prop으로 내려갈땐 () 쓰면 실행되어버리니 받은 compo에서 재정의 해줘야함
 const handleDelete = (id) => { //이게 계속 prop으로 전해져 내려가는 함수
 console.log(11)
 const newItems = items.filter((item)=>{ //item 중 받은 id가 아닌거로만 items 꾸밈
 return item.id !== id
 })
 setItems(newItems) //새 정의된 배열로 setItems
 }

 return (
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>

```

```

 <ReviewList items={sortedItem} onDelete={handleDelete}/> //ReviewList에 handleDelete 넘김
 </div>
)
}

export default App;

import "./ReviewList.css"

function ReviewListItem ({item, onDelete}) { //ReviewList에서 받은 onDelete 여기서 handle~로 정의함
 const handleDeleteClick = () => onDelete(item.id) //파라미터 (id) 받아와야하기 때문
 return (
 <div>

 <div>
 <p>id : {item.id}</p>
 <p>rating : {item.rating}</p>
 <p>{item.content}</p>
 <button onClick={handleDeleteClick}>삭제</button>
 </div>
 <hr><hr>
 </div>
)
}

function ReviewList ({items, onDelete}) { //App에서 받은 함수 다시 ReviewListItem으로 넘겨줌
 return (
 <div>

 {items.map((item, index) => {
 return <li key={index}>{<ReviewListItem item={item} onDelete={onDelete}/>}
 })}

 </div>
)
}

export default ReviewList;

```

- 배열이 바뀔 때마다 재렌더링을 해야하니까 배열을 저장하는데 State를 사용

- 배열을 렌더링 할땐 꼭 key를 지정해줘야
  - 안쓰면 에러메시지 나긴 하지만 작동은 잘됨
  - 안쓰면 배열 값이 삭제되는 등 달라질 때 input 태그를 쓴다 하면 쓴 값이 영뚱한 곳으로 가게 됨
  - (배열의 변화를 리액트에 정확히 전달하는 것)
  - key 값
    - 배열의 index는 그때그때의 배열의 순서이므로 key값으로 쓰면 안됨, id로 사용해야함 (데이터의 부여된 고유값)

```

function ReviewList ({items, onDelete}) {
 return (
 <div>

 {items.map((item) => {
 return <li key={item.id}>{<ReviewListItem item={item} onDelete={onDelete}/>}
 })}

 </div>
)
}

```

- fetch로 데이터 가져오기
  - 이젠 네트워크 통해 fetch로 데이터 가져오기

```

export async function getReviews() {
 const response = await fetch('https://learn.codeit.kr/1636/film-reviews/')
 const body = await response.json();
 return body; //결국 저 링크의 json 파일 리턴
}

import ReviewList from "./ReviewList";

```

```

import mockItems from "../mock.json"
import { useState } from "react";
import { getReviews } from "../api";

function App() {
 const [sort, setSort] = useState("id")
 const [items, setItems] = useState([]) //처음엔 데이터 안뜨게
 const sortedItem = items.sort((a,b) => b[sort]-a[sort])

 const handleRatingSort = () => {
 setSort("rating")
 }

 const handleIdSort = () => {
 setSort("id")
 }

 //{} 이유는 json 안에 reviews라는 객체 안에 배열이 있기 때문
 const handleJson = async() => { //비동기 함수임 표시
 const {reviews} = await getReviews(); //promise 객체 리턴 코드임 표시
 setItems(reviews)
 }

 const handleDelete = (id) => {
 console.log(11)
 const newItems = items.filter((item)=>{
 return item.id !== id
 })
 setItems(newItems)
 }

 return (
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>
 <button onClick={handleJson}>불러오기</button>
 <ReviewList items={sortedItem} onDelete={handleDelete}/>
 </div>
);
}

export default App;

```

- useEffect

- 이전처럼 비동기 함수 - 렌더 무한루프 문제
- 콜백함수 맨 처음 렌더링 할때만 실행
- [dep list] - 이게 변경 시에만 다시 렌더링

```

const handleJson = async() => {
 const {reviews} = await getReviews();
 setItems(reviews)
}

useEffect(()=>{
 handleJson();
},[])

```

- item.sort is not function

- item 배열이 원소가 돌릴만큼 없어서 그런것임

- Cors 정책?

- localhost로 하지마라

- 서버에서 정렬된 데이터 가져오기

- useEffect로 정렬값이 바뀔때마다(버튼) req 해서 서버에서 데이터 받아오기
- 서버에서 데이터 받아오는 이유

- 데이터 개수 많으면 42개 중 10개 정도만 받아오는 식. 이 가져온 랜덤 10개에서 정렬하는게 아니라 42개 중 탑 10개 가져오려면 서버에서 정렬해줘야

- 쿼리

- <https://learn.codeit.kr/1636/film-reviews?order=rating>

저 쿼리는 정해진게 아니라 이 코드잇 서버 만들때 이렇게 구현했음, 약속임

```
const handleJson = async(orderQuery) => { //현재 무슨 order 버튼
 const {reviews} = await getReviews(orderQuery); //order 형식 api에 넘겨줌(쿼리)
 setItems(reviews)
}

//데이터 불러오기는 처음과 order가 변경되었을 때만!
useEffect(()=>{
 handleJson(order);
}, [order])

export async function getReviews(order='createdAt') {
 const query = `order=${order}`
 const response = await fetch(`https://learn.codeit.kr/1636/film-reviews?${query}`)
 const body = await response.json();
 return body;
}
//handleJson에서 받은 order 형식을 서버에 요청하는 url의 쿼리로 넘김
```

- 페이지네이션

- 모든 페이지 한번에 렌더링이 아닌 데이터를 나눠서 가져오는 것
  - 오프셋 = 지금껏 받아온 데이터 개수 (개수 기반)  
기존 데이터에서 추가 or 삭제 시 중복이나 놓치는 데이터 문제
  - 커서 = 특정 데이터 가리킴, 지금껏 받은 거 책갈피  
커서 데이터 이후 10개 가져와줘

```
import ReviewList from "./ReviewList";
import mockItems from "../mock.json"
import { useEffect, useState } from "react";
import { getReviews } from "../api";

const LIMIT = 6; //상수

function App({}) {
 const [order, setOrder] = useState("id")
 const [items, setItems] = useState([])
 const [hasNext, setHasNext] = useState(false) //
 const [offset, setOffset] = useState(0) //
 const sortedItem = items.sort((a,b) => b[order]-a[order])

 const handleRatingSort = () => {
 setOrder("rating")
 }

 const handleIdSort = () => {
 setOrder("id")
 }

 const handleLoad = async(options) => { //현재 무슨 order 버튼
 const {reviews, paging} = await getReviews(options); //이 함수 api에 review, paging 프로퍼티 존재
 if(options.offset == 0){ //처음엔 0~5
 setItems(reviews)
 }
 else{ //이전까 + reviews
 setItems([...items, ...reviews])
 }
 setOffset(options.offset + options.limit) //offset = 기존+6 최신화
 setHasNext(paging.hasNext) //이다음 데이터있는지 paging에서 hasNext 확인
 }
 //api response에 paging과 reviews 프로퍼티 존재

 const handleLoadMore = async() => { //버튼 누르면 실행
 await handleLoad({order, offset, limit:LIMIT}) //지정 offset~6개 불러옴
 }
}
```

```

useEffect(()=>{
 handleLoad({order, offset:0, limit:LIMIT});
}, [order])

const handleDelete = (id) => {
 const newItems = items.filter((item)=>{
 return item.id !== id
 })
 setItems(newItems)
}

return (
 <div>
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>
 </div>
 <ReviewList items={sortedItem} onDelete={handleDelete}/>
 {hasNext && <button disabled={!hasNext} onClick={handleLoadMore}>더 보기</button>}
 </div>
); // hasNext 있으면 button 렌더링 해라 (or hasNext 아니면 버튼 투명화)
}

export default App;

```

- 조건부 렌더링

`{hasNext && <button disabled={!hasNext} onClick={handleLoadMore}>더 보기</button>}` hasNext true면 버튼 렌더링해라  
`{toggle ? <p>✓</p> : <p>✗</p>}` 이런식으로도 가능

- 비동기로 state 변경시 주의

`setItems([...items, ...reviews])`  
`setItems((prevItems) => [...prevItems, ...reviews])`로 변경 시 해결

- 네트워크 스로틀링 해서 느려질때 resp 받는 도중 삭제하면 resp 도착 시 삭제된거 반영이 안됨  
 "함수를 하나의 변수처럼 넘겨준다"

- 네트워크 로딩 처리

- 스로틀링 시 로딩 시 다른거 못하게

```

import ReviewList from "./ReviewList";
import mockItems from "../mock.json"
import { useEffect, useState } from "react";
import { getReviews } from "../api";

const LIMIT = 6;

function App() {
 const [order, setOrder] = useState("id")
 const [items, setItems] = useState([])
 const [hasNext, setHasNext] = useState(false)
 const [offset, setOffset] = useState(0)
 const [isLoading, setIsLoading] = useState(false) //로딩상태 표현할 state
 const sortedItem = items.sort((a,b) => b[order]-a[order])

 const handleRatingSort = () => {
 setOrder("rating")
 }

 const handleIdSort = () => {
 setOrder("id")
 }

 //try문 안에 json 받아오는 것과 그동안 setLoading 참으로 설정
 const handleLoad = async(options) => {
 let result; //getReviews 만 빼어놓기 위해 사용
 try {
 setIsLoading(true)
 result = await getReviews(options);
 } catch(e) {
 console.log(e)
 return;
 } finally {
 setIsLoading(false) //어쨌든 resp 받거나 에러뜨거나 로딩은 끝
 }
 const {reviews, paging} = result
 if(options.offset == 0){

```

```

 setItems(reviews)
 }
 else{
 setItems((prevItems) => [...items, ...reviews])
 }
 setOffset(options.offset + options.limit)
 setHasNext(paging.hasNext)
}

const handleLoadMore = async() => {
 await handleLoad({order, offset, limit:LIMIT})
}

useEffect(()=>{
 handleLoad({order, offset:0, limit:LIMIT});
}, [order])

const handleDelete = (id) => {
 const newItems = items.filter((item)=>{
 return item.id !== id
 })
 setItems(newItems)
}

return (
 <div>
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>
 </div>
 <ReviewList items={sortedItem} onDelete={handleDelete}/>
 {hasNext && <button disabled={isLoading} onClick={handleLoadMore}>더 보기</button>}
 </div> //로딩중이면 버튼 투명화
);
}

export default App;

```

- 네트워크 에러 처리

- 에러가 생겼을 때 catch에서 받은 에러메시지 화면에 띄우기

```

function App() {
 const [order, setOrder] = useState("id")
 const [items, setItems] = useState([])
 const [hasNext, setHasNext] = useState(false)
 const [offset, setOffset] = useState(0)
 const [isLoading, setIsLoading] = useState(false)
 const [loadingError, setLoadingError] = useState(null) //에러 메시지 담을 state
 const sortedItem = items.sort((a,b) => b[order]-a[order])

 const handleRatingSort = () => {
 setOrder("rating")
 }

 const handleIdSort = () => {
 setOrder("id")
 }

 const handleLoad = async(options) => {
 let result;
 try {
 setIsLoading(true)
 setLoadingError(null) //일단 null로 처리
 result = await getReviews(options);
 } catch(e) {
 console.log(e)
 setLoadingError(e) //에러 생기면 담음
 return;
 } finally {
 setIsLoading(false)
 }
 const {reviews, paging} = result
 if(options.offset == 0){
 setItems(reviews)
 }
 else{
 setItems((prevItems) => [...items, ...reviews])
 }
 setOffset(options.offset + options.limit)
 setHasNext(paging.hasNext)
 }
}

```

```

 const handleLoadMore = async() => {
 await handleLoad({order, offset, limit:LIMIT})
 }

 useEffect(()=>{
 handleLoad({order, offset:0, limit:LIMIT});
 }, [order])

 const handleDelete = (id) => {
 const newItems = items.filter((item)=>{
 return item.id !== id
 })
 setItems(newItems)
 }

 return (
 <div>
 <div>
 <button onClick={handleRatingSort}>rating</button>
 <button onClick={handleIdSort}>id</button>
 </div>
 <ReviewList items={sortedItem} onDelete={handleDelete}/>
 {hasNext && <button disabled={isLoading} onClick={handleLoadMore}>더 보기</button>}
 {loadingError?.message && {loadingError.message}}
 </div>
); //loadingError 메시지가 있으면 (catch에서 에러 메시지 받았으면) span 렌더링
 }
}

```

#### ■ 익서널 체이닝 loadingError?.message

원편의 프로퍼티 값이 `undefined` 또는 `null` 이 아니라면 그다음 프로퍼티 값을 리턴하고, 그렇지 않은 경우에는 `undefined`를 반환

푸드잇 (왠만하면 영화쪽에 설명 적어놨으니 영화쪽으로 볼 것)

- jsx에서 map 함수

방법1. `items.map((item) => { return <li><FoodListItem item={item} /></li>} )`

방법2. `items.map((item) => (<li><FoodListItem item={item} /> </li>))` (생략형)

- 음식 목록 보여주기

```

//이게 음식 하나마다 표시할 정보 한 묶음
//이건 아래 foodlist에서 쓰이고 말 것, export는 어차피 아래만됨
//이건 별도 컴포넌트 사실 다른 파일에서 해도되지만 짧고 한눈에 보기 쉽게 한페이지에 씀
function FoodListItem({ item }) {
 const { imgUrl, title, calorie, content } = item;

 return (
 <div>

 <div>{title}</div>
 <div>{calorie}</div>
 <div>{content}</div>
 </div>
);
}

//음식 데이터 전체를 map으로 하나씩 추출해서 하나하나마다 FoodListItem 호출해서 list 하나씩 리턴
function FoodList({ items }) {
 return
 {items.map((item) => { //전체를 한 음식마다로 분해
 return <FoodListItem item={item} />
 })}

}

export default FoodList;

--- map 이렇게도 가능
{items.map((item) => (
 <FoodListItem item={item} />
))}


```

- 정렬

```

import FoodList from './FoodList';
import items from '../mock.json';
import {useState} from "react"

function App() {
 const [sort, setSort] = useState("createdAt")
 const sortedItem = items.sort((a,b) => b[sort]-a[sort])
 const handleCreatedSort = () => {
 setSort("createdAt")
 }
 const handleCalorieSort = () => {
 setSort("calorie")
 }
 return (
 <div>
 <button onClick={handleCreatedSort}>최신순</button>
 <button onClick={handleCalorieSort}>칼로리순</button>
 <FoodList items={sortedItem} />
 </div>
);
}

export default App;

```

- 삭제

```

import { useState } from 'react';
import FoodList from './FoodList';
import mockItems from '../mock.json';

function App() {
 const [order, setOrder] = useState('createdAt');
 const [items, setItems] = useState(mockItems)

 const handleNewestClick = () => setOrder('createdAt');

 const handleCalorieClick = () => setOrder('calorie');

 const sortedItems = items.sort((a, b) => b[order] - a[order]);

 const handleDelete = (id) => {
 const newItems = items.filter((item) => item.id!==id)
 setItems(newItems)
 }

 return (
 <div>
 <button onClick={handleNewestClick}>최신순</button>
 <button onClick={handleCalorieClick}>칼로리순</button>
 <FoodList items={sortedItems} onDelete={handleDelete} />
 </div>
);
}

export default App;

import './FoodList.css';

function formatDate(value) {
 const date = new Date(value);
 return `${date.getFullYear()}. ${date.getMonth() + 1}. ${date.getDate()}`;
}

function FoodListItem({ item, onDelete }) {
 const { imgUrl, title, calorie, content, createdAt } = item;
 const handleDeleteButton = () => {
 onDelete(item.id)
 }

 return (
 <div className="FoodListItem">

 <div>{title}</div>
 <div>{calorie}</div>
 <div>{content}</div>
 <div>{formatDate(createdAt)}</div>

```

```

 <button onClick={handleDeleteButton}>삭제</button>
 </div>
);
}

function FoodList({ items, onDelete }) {
 return (
 <ul className="FoodList">
 {items.map((item) => (

 <FoodListItem item={item} onDelete={onDelete} />

))}

);
}

export default FoodList;

```

- fetch로 데이터 가져오기

```

const handleLoadClick = async() => {
 // 여기에 코드를 작성하세요
 const {foods} = await getFoods();
 setItems(foods)
};

export async function getFoods() {
 const response = await fetch('https://learn.codeit.kr/4366/foods')
 const body = await response.json();
 return body
}

```

- 서버에서 정렬

- 페이지네이션 cursor 방법 & handleload 완벽 이해

```

import { useEffect, useState } from 'react';
import { getFoods } from '../api';
import FoodList from './FoodList';

function App() {
 const [order, setOrder] = useState('createdAt');
 const [items, setItems] = useState([]);
 const [cursor, setCursor] = useState(null); // 커서 처음
 const [isFinished, setIsFinished] = useState(false) // 끝났을때 true로 할 것

 const handleNewestClick = () => setOrder('createdAt');

 const handleCalorieClick = () => setOrder('calorie');

 const handleDelete = (id) => {
 const nextItems = items.filter((item) => item.id !== id);
 setItems(nextItems);
 };

 // nextCursor가 없어도 마지막 페이지는 render 해야지(주의), 그래서 현재 cursor 기준으로 함
 // cursor 없으면 (처음임) 전체 렌더
 // cursor 있으면 (더보기겠지) 이전 데이터 + cursor 이용 새 데이터
 // 쿼리 보낸거 받아오고 그걸 cursor로 할당
 const handleLoad = async (orderQuery) => {
 const { foods, paging } = await getFoods(orderQuery);
 if(!orderQuery.cursor) { // 처음 렌더링 시 - 커서 없을테니
 setItems(foods);
 } else { // 더보기 눌렀을 때
 setItems([...prevItems, ...foods]);
 }
 setCursor(paging.nextCursor);
 };

 // 더보기 버튼 누를 때
 const handleLoadMore = () => {
 handleLoad({order, cursor}); // 더 받을 땐 cursor 이용 / 받는 함수 파라미터가 orderQuery (객체 이므로)
 };

 const sortedItems = items.sort((a, b) => b[order] - a[order]);
}

```

```

//처음과 order 바꿀 때
useEffect(() => {
 handleLoad({order}); //첨엔 cursor 필요 없지
}, [order]);

return (
 <div>
 <button onClick={handleNewestClick}>최신순</button>
 <button onClick={handleCalorieClick}>칼로리순</button>
 <FoodList items={sortedItems} onDelete={handleDelete} />
 <button disabled={!cursor} onClick={handleLoadMore}>더보기</button>
 </div>
); //현 cursor가 없으면 (nextcursor로 할당받을 때 null) 더보기 금지
}

export default App;

```

- 네트워크 slow 시 더보기 버튼 비활 & 에러 시(api에서 에러 던짐) 에러메시지

```

import { useEffect, useState } from 'react';
import { getFoods } from '../api';
import FoodList from './FoodList';

function App() {
 const [order, setOrder] = useState('createdAt');
 const [cursor, setCursor] = useState(null);
 const [items, setItems] = useState([]);
 const [isLoading, setIsLoading] = useState(false) //
 const [loadingError, setLoadingError] = useState("") //

 const handleNewestClick = () => setOrder('createdAt');

 const handleCalorieClick = () => setOrder('calorie');

 const handleDelete = (id) => {
 const nextItems = items.filter((item) => item.id !== id);
 setItems(nextItems);
 };

 const handleLoad = async (options) => {
 let result; //이렇게 선언해서 나눈건 try문 블록 안에서 선언하면 아래서 사용 불가 undef
 try {
 setLoadingError(null)
 setIsLoading(true) //이게 위에 있어야 먼저 버튼이 비활되고 불러오기 시작하지
 result = await getFoods(options);
 } catch (e) {
 setLoadingError(e) //e를 객체로 받아 아래에서 띄워주려고
 return
 } finally {
 setIsLoading(false) //
 }
 //
 const { foods, paging: { nextCursor } } = result;
 if (!options.cursor) {
 setItems(foods);
 } else {
 setItems([...prevItems, ...foods]);
 }
 setCursor(nextCursor);
 };

 const handleLoadMore = () => {
 handleLoad({
 order,
 cursor,
 });
 };

 const sortedItems = items.sort((a, b) => b[order] - a[order]);

 useEffect(() => {
 handleLoad({
 order,
 });
 }, [order]);

 return (
 <div>
 <button onClick={handleNewestClick}>최신순</button>
 <button onClick={handleCalorieClick}>칼로리순</button>
 <FoodList items={sortedItems} onDelete={handleDelete} />
 {cursor && <button disabled={isLoading} onClick={handleLoadMore}>더보기</button>}
 </div>
);
}

export default App;

```

```

 {loadingError?.message && {loadingError.message} }
 </div>
); //e 객체 안에 있는 message 프로퍼티
//loadingError가 있는지 먼저 확인 후 그리고 message가 있는지 2번 확인하는 느낌?
}

export default App;

catch에 return이 있어도 그 직전에 finally 와서 코드 실행하고 return으로 돌아감
finally 이후 코드 = (에러 시 catch에서 return만 하지 않는 이상 실행)


```

## 입력 폼 다루기

- 바닐라 js에선
  - input 입력할때마다 onInput
  - onchange는 인풋 입력 끝냈을때
- 리액트에서의 onchange 대체 - js에서 oninput 처럼 입력할 때 마다 작동됨
- 인풋값을 반영할 state를 만들고 value는 state , onchange는 state를 변경하는 함수 적용

```

const [title, setTitle] = useState("")

const handleTitleChange = (e) => { //변할때마다 title로 state 변경
 setTitle(e.target.value)
}

return (
 <form>
 <input value={title} onChange={handleTitleChange} />
 </form>
);

```

- onSubmit
  - 버튼
    - type="submit"
  - form
    - onSubmit={handleSubmit}
  - handleSubmit에서 preventDefault()

```

import { useState } from "react";
import "./ReviewForm.css"

function ReviewForm() {

 const [title, setTitle] = useState("");
 const [rating, setRating] = useState(0);
 const [content, setContent] = useState('');

 const handleTitleChange = (e) => {
 setTitle(e.target.value)
 }

 const handleRatingChange = (e) => {
 const nextRating = Number(e.target.value);
 setRating(nextRating);
 };

 const handleContentChange = (e) => {
 setContent(e.target.value);
 };

 const handleSubmit = (e) => {

```

```

 e.preventDefault() // 원래 바로 전송되는 기본 성질 (페이지 초기화) 막기 위함
 console.log({title, rating, content}) //원래 네트워크로 전송
 }

 return (
 <form className="ReviewForm" onSubmit={handleSubmit}> //버튼이 존재하는 form 태그에서 onsubmit으로 받음
 <input value={title} onChange={handleTitleChange} />
 <input type="number" value={rating} onChange={handleRatingChange} />
 <textarea value={content} onChange={handleContentChange} />
 <button type="submit">전송</button> //버튼에서는 이벤트 안받고 type만 지정
 </form>
);
}

export default ReviewForm;

```

- 하나의 state와 하나의 handle 함수로 form 구현

- input 태그의 name을 활용

```

import { useState } from "react";
import "./ReviewForm.css"

function ReviewForm() {

 const [values, setValues] = useState({
 title : "",
 rating : 0,
 content : ""
 }) //values라는 객체를 state로 설정

 const handleChange = (e) => {
 const {name, value} = e.target; //target은 input 태그였고, 그 안에 프로퍼티 name, value만 갖고 오겠다
 //위는 방금 변경된 데이터
 setValues({...values, [name]: value}) //변경 안된건 values에, 방금 이벤트 된 value를 추가
 }

 const handleSubmit = (e) => {
 e.preventDefault() // 원래 바로 전송되는 기본 성질 (페이지 초기화) 막기 위함
 console.log(values)
 }

 return (
 <form className="ReviewForm" onSubmit={handleSubmit}>
 <input name="title" value={values.title} onChange={handleChange} />
 <input type="number" name="rating" value={values.rating} onChange={handleChange} />
 <textarea name="content" value={values.content} onChange={handleChange} />
 <button type="submit">전송</button>
 </form>
);
}

export default ReviewForm;

```

- 문법 주의

■ `setValues({...values, [name]: value})`

이려면 name을 프로퍼티로 사용 가능하다

- 객체도 spread 가능

```

const [values, setvalues] = useState({id: '123', pw: 123});
...
{name, value} = e.target;
setValues({...values, [name]: value});

```

- 제어 비제어 컴포넌트

- input 값을 리액트가 지정 (제어)

항상 리액트 사용 값 = 실제 인풋 값

```
<input value={value} onChange={ 이 함수에서 setValue } >
```

value가 있고 없고 차이임

- 있으면 onChange에서 변형한 결과로 리액트에 저장 및 인풋에도 그렇게 쓰임

- 없으면 리액트에만 그렇게 저장, 인풋엔 쓴 그대로 보임

값을 예측하기가 쉽고 인풋에 쓰는 값을 여러 군데서 쉽게 바꿀 수 있다는 장점

- 기본 파일 input 구조

```
import { useState } from "react"

function FileInput () {
 const [file, setFile] = useState()

 const handleFile = (e) => {
 console.log(e.target.files) //파일 리스트 객체 출력
 }

 return (<input type="file" value={file} onChange={handleFile}></input>) //input 태그 하나 주는 것
} //type=file 만 적어도 업로드 버튼 형식 갖춰짐 / onChange로 파일 올린 뒤 컨트롤

export default FileInput
```

- FileList 객체 0번째 프로퍼티 = 네트워크 전송 및 이미지 미리보기 등 사용하게 함

- 파일 인풋은 target.files를 사용

- 파일 인풋은 비제어 인풋으로 만들어야

- 인풋은 보안 상 사용자만 바꿀 수 있어야

- 사용자 파일 경로 숨겨줌 - fakepath

- 오류남 - `value={file}` 지워줘야

- 중간 (아래 lifting으로 가기 이전 코드)

```
import { useState } from "react";
import FileInput from "./FileInput";
import "./ReviewForm.css"

function ReviewForm() {

 const [values, setValues] = useState({
 title : "",
 rating : 0,
 content : ""
 })

 const handleChange = (e) => {
 const {name, value} = e.target;
 setValues({...values, [name]: value})
 }

 const handleSubmit = (e) => {
 e.preventDefault()
 console.log(values)
 }

 return (
 <form className="ReviewForm" onSubmit={handleSubmit}>
 <FileInput />
 <input name="title" value={values.title} onChange={handleChange} />
 <input type="number" name="rating" value={values.rating} onChange={handleChange} />
 <textarea name="content" value={values.content} onChange={handleChange} />
 <button type="submit">전송</button>
 </form>
);
}

export default ReviewForm;
```

```

import { useState } from "react"

function FileInput () {
 const [file, setFile] = useState()

 const handleFile = (e) => {
 const newFile = e.target.files[0]
 setFile(newFile)
 }

 return (<input type="file" onChange={handleFile}></input>) //input 태그 하나 주는 것
} //type="file" 만 적어도 업로드 버튼 형식 갖춰짐 / onChange로 파일 올린 뒤 컨트롤

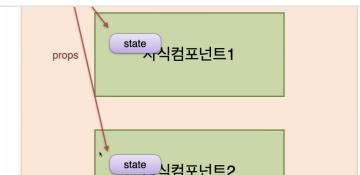
export default FileInput

```

- file 컴포넌트에서 state 대신 prop을 사용 = state lifting

- 자식 컴포넌트의 스테이트를 부모 컴포넌트로 옮겨주는 것을 말한다.

그러면 자식 컴포넌트가 state 공유 가능



```

state Lifting(스테이트 끌어올리기)
리액트에서는 컴포넌트를 나눠서 사용한다. 이때 부모컴포넌트에 있는 데이터를 자식컴포넌트에서 사용하고 싶다면 props를 이용해 데이터를 주고 받을 수 있다. 하지만 부모와 자식간에 데이터 이동은 가능하지만 자식과 자식 컴포넌트끼리는 데이터를 주고 받을 수 없다. 만약 다수의
https://velog.io/@uiop5487/state-Lifting

```

```

import { useState } from 'react';
import FileInput from './FileInput';
import './ReviewForm.css';

function ReviewForm() {
 const [values, setValues] = useState({
 title: '',
 rating: 0,
 content: '',
 imgFile: null,
 });

 const handleChange = (name, value) => { //파일은 이거만 사용
 setValues((prevValues) => ({
 ...prevValues,
 [name]: value,
 }));
 };

 const handleInputChange = (e) => { //인풋들은 이거랑 위에도 사용
 const { name, value } = e.target;
 handleChange(name, value);
 };

 const handleSubmit = (e) => {
 e.preventDefault();
 console.log(values);
 };

 return (
 <form className="ReviewForm" onSubmit={handleSubmit}>
 <FileInput name="imgFile" value={values.imgFile} onChange={handleChange} />
 <input name="title" value={values.title} onChange={handleInputChange} />
 <input type="number" name="rating" value={values.rating} onChange={handleInputChange} />
 <textarea name="content" value={values.content} onChange={handleInputChange} />
 <button type="submit">확인</button>
 </form>
);
}

export default ReviewForm;

```

```

function FileInput({ name, value, onChange }) { //value는 후에 미리보기 등으로 활용 가능해서 일단 받아놓음..

 const handleChange = (e) => {
 const nextValue = e.target.files[0];
 onChange(name, nextValue); //부모의 것 실행 (prop으로 받음)

```

```

};

return <input type="file" onChange={handleChange} />;
} //input 값 바뀌면 위 함수 호출, 여기서 onChange는 위 받은 것과 별개

export default FileInput;

```

▪ 일반적 prop과 비교하고 해야하는데 좀 아리끼리함 - 나중에 꼭 다시 봐야될 중요한 내용!

- `FileInput.js` 의 `onChange(name, nextValue)` 에서 `onChange` 는 `ReviewForm.js`에서 넘어온 `handleChange` 함수입니다.

즉 위 `handleChange`와 아래 `handleChange`는 아예 다른 함수

▼ 심플 예제로 완벽한 설명

```

App.js
import {useState} from 'react';

function MyComponent({value, onChange}) {
 const handleChange = ((e) => {
 const nextValue = e.target.value.toUpperCase();
 onChange(nextValue);
 });
 return <input value={value} onChange={handleChange}/>;
}

function App() {
 const [value, setValue] = useState("");
 const handleClear = () => setValue("");
 return (
 <div>
 <MyComponent value={value} onChange={setValue}>/>
 <button onClick={handleClear}>지우기</button>
 </div>
);
}

export default App;

```

이것은 앞에 나왔던 내용인데요. 거기서 하나 빠먹은 것이 있습니다.

`App` 컴포넌트에서, `MyComponent`에게 `value` 와 `onChange` props를 넘겨주었는데,

그로인해, 하위 컴포넌트인 `MyComponent`에서의 `onChange`는 `setValue` 함수로 대체가 되는데,

문제는 `setValue` 함수는, 상위컴포넌트에 속해있는 함수입니다.

하위컴포넌트인 `MyComponent`에서는 `setValue` 함수가 정의된 적이 없습니다.

그런데, 특이하게도, 하위컴포넌트에서, 상위컴포넌트의 함수를 호출해서 사용합니다.(보라색화살표)

이것을 보면, 상위컴포넌트에서 `onChange={setValue}` 라고 할 때,

단순히 `setValue`라는 이름만 넘겨주는 것이 아니라, `setValue` 함수가 메모리상에 할당된 주소를 같이 넘겨주는 것 같습니다.

그렇기 때문에, 하위컴포넌트에서도, 상위컴포넌트의 함수를 참조해서 실행할 수 있는 것이겠죠.

자. 그럼 생각을 해 봅시다.

왜 이런 방식을 써야 되는 걸까?

우선, 하위컴포넌트 `MyComponent`는 `input` 태그를 만들고, `input`으로 부터 '데이터'(value)를 받아들입니다.

그러면, 그 데이터가, 상위컴포넌트에 있는 `value`라는 `state`를 업데이트해야 되는데,

그걸 업데이트 하려면, 상위컴포넌트에 있는 `setValue` 함수를 호출해야 됩니다.

그러므로, 상위컴포넌트에서는 `onChange` prop에 `setValue` 함수를 실어서 보내는 것이죠.

`onChange={setValue}` 외에

value 값도 같이 보냈는데, 이것은 `input` 태그에서 `value={value}`에서 표시할 기본값(default값)으로 사용하기 위한 것 `onChange`는 `setValue` 함수를 받아서 \*\*실행준비상태\*\*에 놓입니다. 이게 실행되려면, `handleChange` 함수가 실행(발동)되어야 됩니다.

`handleChange` 함수는,

`input` 자극(e)에 의해서 (`input` 태그에 있는) `onChange` 이벤트로 발생하는, 이벤트핸들러입니다.

그래서 `input`에 자료가 입력되면, 새로운 value가 발생(`e.target.value`)하고,

순식간에 `onChange`(=`setValue`)함수가 작동해서, 상위컴포넌트의 `value`를 바꿔버리니,

소문자를 입력해도, 대문자를 입력하고 있는 것처럼 느껴지게 만듭니다.(거의 동시적이라서)

`input` 태그에 `value={value}`라고 할 때, `value`는 기존에 있던 `value`를 보여주는 것인데...

너무 동시적으로 `setValue` 함수가 작동해서 `value`값을 바꿔버리니까, 내가 입력하고 있는 값이, 그래도 보여지게 되는 거죠.

#### ▼ 위 예제로 완벽한 설명

The diagram illustrates the prop flow between two components: `FileInput.js` and `ReviewForm.js`.

`FileInput.js` code:

```
function FileInput({ name, value, onChange }) {
 const handleChange = (e) => {
 const nextValue = e.target.files[0];
 onChange(name, nextValue);
 };
 return <input type="file" onChange={handleChange} />;
}
export default FileInput;
```

`ReviewForm.js` code:

```
import { useState } from "react";
import "./ReviewForm.css";

function ReviewForm() {
 const [values, setValues] = useState({
 title: "", rating: 0, content: "", imgFile: null,
 });
 const handleChange = ({ name, value }) => {
 setValues((prevValues) => ({ ...prevValues, [name]: value }));
 };
 const handleInputChange = (e) => {
 const { name, value } = e.target;
 handleChange(name, value);
 };
 const handleSubmit = (e) => {
 e.preventDefault();
 console.log({ title, rating, content });
 };
 return (
 <form className="ReviewForm" onSubmit={handleSubmit}>
 <Fileinput
 name="imgFile"
 value={values.imgFile}
 onChange={handleChange}
 />
 <input value={values.title} onChange={handleInputChange} />
 <input type="number" value={values.rating} onChange={handleInputChange} />
 <textarea value={values.content} onChange={handleInputChange} />
 <button type="submit">확인</button>
 </form>
);
}
export default ReviewForm;
```

An orange arrow points from the `onChange` prop in the `FileInput` component to the `handleChange` function in the `ReviewForm` component. Another orange arrow points from the `Fileinput` component to the `ReviewForm` component, indicating the flow of the entire prop object.

앞의 예제는 하나의 파일(`App.js`)안에서 작동했던 것이고,

이번 내용은 지금 해당 수업내용으로, 하위컴포넌트 파일인 `FileInput.js` 과 상위컴포넌트인 `ReviewForm.js` 간에 props 이동에 관한 것입니다.

`ReviewForm` 컴포넌트에서 `FileInput` 컴포넌트로 `name`, `value`, `onChange` props를 내려주었습니다.

`onChange` prop에는 상위컴포넌트의 함수인 `handleChange` 함수를 담아서 보내주었습니다.

그리고, `handleChange` 함수에 들어갈 인자인, `name`, `value` 도 같이 보내주었습니다.

하위컴포넌트에서는 함수와 인자를 다 받았으니, 잘 실행할 수 있겠죠.

이번 경우도 역시, 하위컴포넌트에서,

`input`된 자료를 바탕으로

상위컴포넌트의 `values state`를 변화시키기 위해서 이렇게 한 것이네요.

위 도표를 다음과 같이 수정할 수 있을 것 같습니다.

```

FileInput.js
function FileInput({ name, value, onChange }) {
 const handleChange = (e) => {
 const nextValue = e.target.files[0];
 onChange(name, nextValue);
 };

 return <input type="file" onChange={handleChange} />;
}

export default FileInput;

ReviewForm.js
import { useState } from "react";
import "./ReviewForm.css";

function ReviewForm() {
 const [values, setValues] = useState({
 title: "", rating: 0, content: "" imgFile: null, });

 const handleChange = ({ name, value }) => {
 setValues((prevValues) => ({ ...prevValues, [name]: value }));
 };
 const handleInputChange = (e) => {
 const { name, value } = e.target;
 handleChange(name, value);
 };
 const handleSubmit = (e) => {
 e.preventDefault();
 console.log({ title, rating, content });
 };

 return (
 <form className="ReviewForm" onSubmit={handleSubmit}>
 <FileInput
 name="imgFile"
 value={values.imgFile}
 onChange={handleChange}>
 />
 <input value={values.title} onChange={handleInputChange} />
 <input type="number" value={values.rating} onChange={handleInputChange} />
 <textarea value={values.content} onChange={handleInputChange} />
 <button type="submit">확인</button>
 </form>
);
}

export default ReviewForm;

```

The diagram illustrates the flow of the `onChange` prop. A blue arrow points from the `ReviewForm` component's `handleChange` function to the `FileInput` component's `handleChange` function. Another blue arrow points from the `ReviewForm` component's `handleInputChange` function to the `FileInput` component's `handleChange` function. An orange arrow points from the `FileInput` component's `handleChange` function back to its own `onChange` prop.

하위컴포넌트에서 다시 상위컴포넌트의 함수를 호출하는 형태가 아니라... (이것은 기본적으로 불가능하다고 생각되고..  
왜냐면, 하위컴포넌트가 상위컴포넌트함수를 import하지 않았으므로)

`onChange` prop이, 하위컴포넌트로, `handleChange` 함수를 실어 보냈다.

이런 느낌 같습니다.

- ref라는 prop
  - 원하는 시점에 실제 dom 노드에 접근하고 싶을 때
  - dom 노드는 렌더링 끝나야 생기니 ref도 렌더링 되었을 때만 존재
    - 그래서 if(!inputNode) 하는거임 (current 값을 확인!)

```

const inputRef = useRef();
const inputNode = inputRef.current
inputNode.value ~

```

```

▼ {current: input} ⓘ
 ▼ current: input
 value: "C:\\fakepath\\\\반명함.jpg"
 ▶ __reactEvents$g7r03epkjr8: Set(1) {'in

```

inputRef 구성요소

- value 속성을 빈 문자열로 설정 ⇒ 파일 초기화

```

import { useEffect, useRef } from "react";

function FileInput({ name, value, onChange }) { //value는 후에 미리보기 등으로 활용 가능해서 일단 받아놓음..
 const inputRef = useRef(); //현재 input 상태?

 const handleChange = (e) => {
 const nextValue = e.target.files[0];
 onChange(name, nextValue); //부모의 것 실행 (prop으로 받음)
 };

 const handleClearClick = () => {
 const inputNode = inputRef.current
 if(inputNode) { //파일이 있다면 value 없애고
 inputNode.value=''; //초기화
 onChange(name, null) //name은 imgtitle (프로퍼티)니까 이게 없다는걸 알려야하므로 적어야지
 }
 }

```

```

 return (
 <div>
 <input type="file" onChange={handleChange} ref={inputRef}/>
 {value && <button onClick={handleClearClick}> delete </button> }
 </div>
); //value는 위 파라미터에서, 현재 파일이 있는지 확인 위함, 있으면 render
 }

export default FileInput;

```

- o `input에 ref={useRef() 반환값}`으로 참조시켜놓고 반환값을 이용해 input 태그 조작 느낌

- 이미지 파일 미리보기

- o `useEffect`에서 `dep`를 `value`로 해서 파일 선택 시마다 실행되도록
- o `objurl`은 `URL.createObjectURL(value)`로 생성 = url string 값 가짐
- o 사이드 이펙트를 사용하는 경우 `useEffect` 사용 = network req나 메모리 할당같이 컴포넌트 함수에서 외부의 상태를 변경할 때

```

import { useEffect, useRef, useState } from "react";

function FileInput({ name, value, onChange }) { //value는 후에 미리보기 등으로 활용 가능해서 일단 받아놓음..
 const inputRef = useRef();
 const [preview, setPreview] = useState(); //이미지 url 담을 변수

 const handleChange = (e) => {
 const nextValue = e.target.files[0];
 onChange(name, nextValue);
 };

 const handleClearClick = () => {
 const inputNode = inputRef.current
 if(inputNode) {
 inputNode.value='';
 onChange(name, null)
 }
 }

 useEffect(()=>{
 if(!value) return; //혹시 파일 없으면 아래 실행시 오류
 setPreview(URL.createObjectURL(value)) //이미지 태그에 적용할 url
 }, [value])

 return (
 <div>
 //src=url
 <input type="file" onChange={handleChange} ref={inputRef}/>
 {value && <button onClick={handleClearClick}> delete </button> }
 </div>
);
}

export default FileInput;

```

- 사이드 이펙트

- o = 파일 선택 시마다 메모리에 할당됨 - 계속하면 메모리 낭비됨
- o `revokeURLobject` - 메모리 정리 (사이드 이펙트 정리)
- o `return`의 콜백함수인 정리함수를 기억해두고, 파일을 선택해 `value` 값이 바뀌어서 재랜더링 시작할 때 먼저 실행됨 (먼저 메모리 정리하고 선택한 파일 이미지 메모리에 올림)

```

useEffect(()=>{
 if(!value) return; //혹시 파일 없으면 아래 실행시 오류
 const nextPreview = URL.createObjectURL(value)
 setPreview(nextPreview) //이미지 태그에 적용할 url
 return () => {
 setPreview() //url 삭제하고
 URL.revokeObjectURL(nextPreview) //메모리에서 삭제
 }
}, [value])

```

- 별점 컴포넌트 추가..

```
import './Rating.css'

function Star({selected = false}) {
 const className = `Rating-star ${selected ? 'selected' : ''}`
 return *
}

function Rating({value=0}) {
 return (
 <div>
 <Star selected={value >= 1} />
 <Star selected={value >= 2} />
 <Star selected={value >= 3} />
 <Star selected={value >= 4} />
 <Star selected={value >= 5} />
 </div>
)
}

export default Rating;

.Rating-star {
 color: slategray
}

.Rating-star.selected {
 color: yellowgreen
}

```

```
const RATING = [1,2,3,4,5]

{RATING.map((rating)=>{
 <Star key={rating} selected={value >= rating}/>
})}
```

- star - Rating - reviewListItem

- ```
useEffect(() => {
const timerId = setInterval(() => console.log('tock'), 1000);
return () => clearInterval(timerId);
}, []);
```