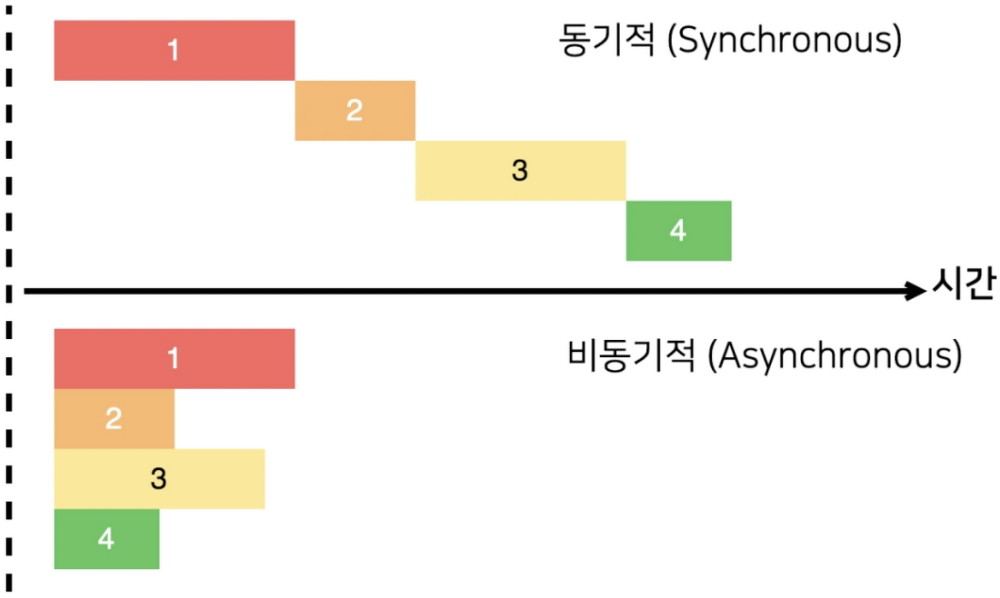
09.06_실습 강의자료

0교시

▼ 동기 비동기 복습!



동기 : 작업이 끝날 때까지 중지 상태가 되어 다른 작업을 할 수 없다.

비동기 : 동시에 여러가지 작업을 처리할 수있다.

JS에도 비동기가 작동하는 방식

JS는 싱글 스레드이기 때문에 한번에 한가지만 작동할 수 있습니다. 하지만 비동기로 여러가지를 처리할 수 있죠. 그 방법은

1. call stack에서 정해진 비동기 함수를 호출하여
2. Web APIs 등 백그라운드에서 작동하는 비동기 함수를 통해 작업을
3. 반환된 결과를 queue에 저장합니다.
4. call stack에 쌓인것을 다 처리 한 후 event loop가 각종 queue를 우선순위별로 순찰하며 하나씩 불러옵니다.

1교시

01. async & await 개념

▼ async await

Promise를 활용해서 비동기 코드를 간결하게 작성하는 문법입니다.

async를 통해 비동기코드 라는 것을 명시해주고 그 안에 await을 사용해서 비동기 처리가 성공했을 경우를 반환 해줍니다.

async 함수는 Promise 객체를 리턴해야하며 Promise가 아닌경우 Promise.resolve()로 감싸집니다. 또 다른 키워드 await는 async 함수 안에서만 동작합니다.

자바스크립트는 await 키워드를 만나면 Promise가 처리될 때까지 기다립니다. 결과는 그 이후 반환됩니다.

예제코드를 보고 이해해보겠습니다.

```
async function f() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("완료!"), 1000);
    });

    let result = await promise; // Promise가 이행될 때까지 기다림
    // let result =  promise; // 안 기다리면 pending상태의 promise 반환

    console.log(result); // "완료!"
}

f();
```

함수를 호출하고, 함수 본문이 실행되는 도중에 await로 표시한 줄에서 실행이 잠시 '중단'되었다가 프로미스가 처리되면 실행이 재개됩니다.

이때 프로미스 객체의 반환값이 변수 result에 할당됩니다. 따라서 위 예시를 실행하면 1초 뒤에 '완료!'가 출력됩니다.

await는 말 그대로 프로미스가 처리될 때까지 함수 실행을 기다리게 만듭니다. 프로미스가 처리되면 그 결과와 함께 실행이 재개되죠.

await는 promise.then보다 좀 더 세련되게 프로미스의 result 값을 얻을 수 있도록 해주는 문법입니다. promise.then보다 가독성 좋고 쓰기도 쉽습니다.

async,await 문법은 try, catch 문을 통해 error처리를 해줄 수 있습니다.

▼ Promise에서 async로!

자 그러면 async await가 어떤 친구인지 대략적으로 감이 오셨다고 생각하고 이전에 배운 promise를 async await로 바꾸는 방법을 알아보겠습니다.

먼저 `Promise`를 반환하는 함수를 `async` 함수로 변경해 줍니다.

아래의 방법으로 바꿀 수 있습니다.

- 함수에 `async` 키워드를 붙입니다.
- `new Promise...` 부분을 없애고 함수본문 내용만 남깁니다.
- `resolve(value);` 부분을 `return value;` 로 변경합니다.
- `reject();` 부분을 `throw` 로 수정합니다.

예제와 함께 보시죠

```
/*promise*/
function startAsync(age) {
```

```
    return new Promise((resolve, reject) => {
        if (age > 20) resolve(`${age} success`);
        else reject(new Error(`${age} is not over 20`));
    });
}

/* async */
async function startAsync(age) {
    if (age > 20) return `${age} success`;
    else throw new Error(`${age} is not over 20`);
}


// 성공시
const promise1 = startAsync(25);
// async func은 항상 promise를 반환합니다.
// console.log(startAsync(25));

promise1
    .then((value) => {
        console.log(value);
    })
    .catch((error) => {
        console.error(error);
    });

// 실패시
const promise2 = startAsync(15);
promise2
    .then((value) => {
        console.log(value);
    })
    .catch((error) => {
        console.error(error);
    });
```

그렇다면 이번에는 `then, catch` 문 대신 `await`를 사용하는 방법을 알아볼까요?

그러기 위해 `await`를 사용하는 제약조건을 알아야 합니다.

`await`는

- `async function`안에서만 사용할 수 있습니다.
- `await [promise 객체]` 문법으로 사용할 수 있습니다.
- `await`는 `promise`가 완료될때까지 기다리고(fulfilled상태) `resolve`한 값을 내놓습니다.
- 해당 `promise`에서 `reject`가 발생하면 예외가 발생합니다.

먼저 `await`는 `async`안에서만 사용할 수 있기 때문에 `then, catch 문`에서 하는 작업을 `async func`으로 감싸줘야 합니다. 이때 `async await`에서는 error처리를 `try, catch문`으로 할 수 있기 때문에 `then, catch 메서드` 대신 `try catch문`으로 작성해 줍니다.

- `async function` 로 감싸줍니다.
- `then, catch`부분을 `try catch` 문으로 바꿔줍니다.
- `async function` 에 해당하는 변수에 `await`문을 작성합니다.
- 감싼 `async function` 를 호출합니다.

```
/* async await */
async function startAsync(age) {
    if (age > 20) return `${age} success`;
    else throw new Error(`${age} is not over 20`);
}

async function startAsyncJobs() {
    //  성공시
    try {
        const promise1 = await startAsync(25);
        console.log(promise1);
```

```
    } catch (e) {
        console.error(e);
    }

    // 실패시
    try {
        const promise2 = await startAsync(15);
        console.log(promise2);
    } catch (e) {
        console.error(e);
    }
}
startAsyncJobs();
```

▼ promise VS async

그러면 async await를 쓰면 promise를 모두 대체 할 수 있냐? 그건 아닙니다.

async await만으로는 해결할 수 없는 사례들을 보겠습니다.

비동기 함수로 가장 기본적으로 사용되는 setTimeout같은 경우 async함수로 만들어줄 수 없습니다.

그 이유는Promise로 구현할 때 setTimeout안에 있는 callback함수로써 resolve(value)를 실행했지만 async에서는 그저 return(value)로 변경해 사용하기 때문입니다.

즉, Promise에서 resolve를 호출하는 것이 우리가 아니라 setTimeout이 자신의 타이머가 끝나면 그때 호출 하기 때문입니다. 따라서 우리가 핸들링할 수가 없습니다.

```
// async로 못 바꿈,,,
function setTimeoutPromise(delay) {
    return new Promise((resolve) => setTimeout(resolve, delay));
}
```

Promise.all은 비동기 코드를 동시에 실행시킬 수 있습니다. 비동기 코드를 멀티 스레드로 사용하는 느낌이죠.

하지만 async는 promise를 간결하게 사용할 수는 있지만 await 키워드에서 prmise가 resolove된 이후 다음 코드로 넘어가기 때문에 비동기 코드들을 싱글 스레드로 사용하는 느낌입니다.

그렇다면 asyc를 병렬처리하기 위해서는 어떻게 해야 할까요?

예제 코드를 통해 알아보겠습니다.

```
function delay(ms) {
    return new Promise((resolve) => {
        setTimeout(resolve, ms);
    });
}

async function getApple() {
    await delay(1000);
    return "apple";
}

async function getBanana() {
    await delay(1000);
    return "banana";
}

/* 직렬 처리 */
async function pickFruits1() {
    const apple = await getApple(); //1초 기다리고 실행
    const banana = await getBanana(); //1초 기다리고 실행

    console.log(`${apple} + ${banana}`);
}
```

```
pickFruits1();

/* 병렬 처리 */
// Promise성질 이용
async function pickFruits2() {
    const applePromise = getApple(); //Promise 객체 안 코드 블록 바로실행
    const bananaPromise = getBanana(); //Promise 객체 안 코드 블록 바로실행
    const apple = await applePromise;
    const banana = await bananaPromise;

    console.log(`${apple} + ${banana}`);
}
pickFruits2();

// Promise.all 이용
async function getFruites3() {
    console.time();
    let [apple, banana] = await Promise.all([getApple(), getBanana()]); // 구조 분해로 각 프로미스 리턴값들을 변수에 담는다.
    console.log(`${apple} + ${banana}`);
}

getFruites3();
```

두 차이점을 알고 모두 사용 하실 수 있다면 더 효율적인 코드를 작성할 수 있을 것 입니다.

2교시

02. 철인 3종 경기

▼ await 유무 확인하기

await을 사용했을 때와 사용하지 않았을 때 어떤 차이가 있을까요?

3교시

03. 유저 정보 유청하여 rendering 하기

▼ CSS 라이브러리 사용하기

html 태그를 보면 css라이브러리를 link해왔습니다. 1주차에 배웠던 CSS 3가지 방법중 외부 파일을 만들어서 link를 통해 가져왔었죠?

동일하게 남이 이미 만들어 놓은 CSS 설정을 가져올 수도 있습니다.

그럼 여기서 integrity란 속성은 어떤 것일까요?

integrity란 진실성, 도덕성 이라는 뜻으로 암호값을 넣으므로써 파일이 변경, 변조되어 악용되는 사례를 막는 것입니다.

integrity 속성은 link, script 코드에서 사용할 수 있으며 실습에 사용된 코드에는 해시 값으로 암호화 해놓았습니다.

이러한 보안은 HTTPS를 통해 거의 예방이 가능합니다.

crossorigin는 링크가 CORS정책을 지원하기 위한 속성으로

CORS는 Cross Origin Resource Sharing의 약자로 한국어 직역은 교차 출처 리소스 공유라는 뜻입니다.

이것을 풀어서 해석하자면 다른 출처로부터 온 리소스를 공유한다 라는 뜻입니다.

CORS : https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

▼ JSON 이란?

• JavaScript Object Notation라는 의미의 축약어로 데이터를 저장하거나 전송할 때 많이 사용되는 경량의 DATA 교환 형식입니다. 사람과 기계 모두 이해하기 쉬우며 용량이 작아서, 최근에는 JSON이 XML을 대체해서 데이터 전송 등에 많이 사용합니다.

자주쓰는 메서드

- **JSON.parse(JSON으로 변환할 문자열)** : JSON 형식의 텍스트를 자바스크립트 객체로 변환한다.

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse
- **JSON.stringify(JSON 문자열로 변환할 값)** : 자바스크립트 객체를 JSON 텍스트로 변환한다.

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify

+

- **Response.json() :** json()결과는 JSON이 아니라 대신 JSON을 입력으로 사용하고 이를 구문 분석하여 JavaScript 개체를 생성한 결과입니다.
  - https://developer.mozilla.org/en-US/docs/Web/API/Response/json

▼ fetch VS Axios

Fetch와 Axios 둘 다 HTTP 요청을 처리하기 위한 자바스크립트의 라이브러리 이지만 몇 가지 차이점이 존재합니다.

1. Fetch의 경우 자바스크립트에 내장되어 있기 때문에 별도의 설치가 필요하지 않습니다. 하지만 Axios의 경우 간단하지만 설치 과정이 필요합니다.
2. Fetch는 일부 예전의 인터넷 익스플로러 버전에서 지원하지 않는 경우가 있어, Axios가 Fetch보다 브라우저 호환성이 뛰어납니다.
3. Fetch에서는 지원하지 않는 JSON 자동 변환, 응답 시간 초과 설정 기능 등을 Axios에서 지원해줍니다.

▼ async await

fetch 함수는 Promise를반환한다고 했었죠? 그래서 then 메서드 로 후속 처리를 해주었습니다.

이번에는 async await을 이용해서 문제를 풀어보겠습니다.

다만 Elice IDE에서는 async await을 사용할 수 없습니다.

그 이유는 JS 컴파일러인 바벨은 ES6이후의 JS를 ES5이전의 JS로 변환해주어 구버전의 웹브라우저도 대응을 해줍니다. 하지만 ES8에서 새로나온 async/await 문법은 이전에 없었던 것이기 때문에 변환을 해주지 못합니다. 그래서 바벨 폴리필(babel/Polyfill)를 통해 최신 JS문법을 쓸 수 있게 설정해줘야 합니다.

하지만 Elice IDE를 우리가 설정 할 수 없으니 VScode에서 async/await 코드를 작성 해보아요🙂

```
//randomuser.me 라는 API는 무작위로 생성된 사용자의 프로필 이미지를 포함해 디테일한 정보까지 어디서든 요청 가능합니다.

document.getElementById("myBtn").addEventListener("click", updatePost);

async function getData(url) {
    return fetch(url)
        .then((res) => res.json())
```

```
        .then((data) => data.results);
}

async function updatePost() {
    const data = await getData("https://randomuser.me/api/?results=100");
    // console.log(response);
    console.log(data);
    // 사용자 정보를 요청합니다.
    let output = "<h2><center>사용자 정보 받기</center></h2>";

    // 2. forEach()를 사용해서 user의 각 데이터를 output에 추가합니다.

    data.forEach((lists) => {
        output += `
                    <div class="container">
                        <div class="card mt-4 bg-light">
                            <ul class="list-group">
                                <li class="list-group-item list-group-item-primary"><h2>Name: ${lists.name.first}</h2></li>
                                <li class="list-group-item"><img src="${lists.picture.large}"></li>
                                <li class="list-group-item">Phone Number: ${lists.cell}</li>
                                <li class="list-group-item">DOB: ${lists.dob.date}</li>
                                <li class="list-group-item">Age: ${lists.dob.age}</li>
                                <li class="list-group-item">Email ID: ${lists.email}</li>
                                <li class="list-group-item">City: ${lists.location.city}</li>
                                <li class="list-group-item">Country: ${lists.location.country}</li>
                                <li class="list-group-item">PostCode: ${lists.location.postcode}</li>
                            </ul>
                        </div>
                    </div>`;
    });
    document.getElementById("output").innerHTML = output;
}
```

이렇게 코드를 변경해주면 then 메서드 안에서만 비동기 처리를 해야하는 불편을 해결하고 await을 통해 비동기 처리가 될때까지 기다렸다가 결과를 변수에 넣어주어 코드흐름을 동기처럼 작성할 수 있습니다 🙂

이것으로 JS의 핵심 개념을 대부분 다루었고 다음주에는 React로 찾아뵙겠습니다 안녕~~🙂

+ 추가

▼ Promise 좀 더 알아보기!

Promise안에 callback함수를 만들 때 고려해야 할 부분이 있습니다.

- `callbackFun` 내부에서 에러가 `throw` 된다면 해당 에러로 `reject` 가 수행됩니다.
- `callbackFun` 의 리턴 값은 무시됩니다.
- 첫 번째 `reject` 혹은 `resolve` 만 유효합니다. (두 번째부터는 무시됩니다. 이미 해당 함수가 호출되었다면 `throw` 또한 무시됩니다.)

```
/* 12.21/plus/promise.js */

// catch 로 연결됩니다.
const throwError = new Promise((resolve, reject) => {
    throw Error("error");
});
throwError
    .then(() => console.log("throwError success"))
    .catch(() => console.log("throwError catched"));
//

// 아무런 영향이 없습니다.
const ret = new Promise((resolve, reject) => {
    return "returned";
});
```

```
ret.then(() => console.log("ret success")).catch(() => console.log("ret catched"));

// resolve 만 됩니다.
const several1 = new Promise((resolve, reject) => {
    resolve();
    reject();
});
several1
    .then(() => console.log("several1 success"))
    .catch(() => console.log("several1 catched"));

// reject 만 됩니다.
const several2 = new Promise((resolve, reject) => {
    reject();
    resolve();
});
several2
    .then(() => console.log("several2 success"))
    .catch(() => console.log("several2 catched"));

// resolve 만 됩니다.
const several3 = new Promise((resolve, reject) => {
    resolve();
    throw new Error("error");
});
several3
    .then(() => console.log("several3 success"))
    .catch(() => console.log("several3 catched"));
```

▼ JS의 비동기 역사

전체적인 흐름인 https://string.tistory.com/114 에서 확인하시길 바랍니다.

우리가 배운 비동기의 역사를 살펴보면

CallBack ⇒ Promises(2015/ES6) ⇒ async/await (2017/ES8)

비동기로 많이 사용하는 HTTP 통신같은 경우는

XMLHTTPRequest(1999년) ⇒ ajax(2004년에 나오긴 했지만 2005년 구글이 구글맵 만들면서 사용하기 전까진 관심밖) ⇒ fetch(2015/ES6) ⇒ axios

axios : https://axios-http.com/kr/

이런식으로 JS의 비동기 역사는 이전 방법의 단점들을 보완해나가며 새로운 문법들이 생겨났습니다. 하지만 그렇다고 새로 생긴 방법에 이전의 방법을 100% 대체하는 것은 아닙니다.

따라서 모두 알고는 있어야겠죠..??ㅎㅎ…