Javascript 기초

비동기 처리

김태민

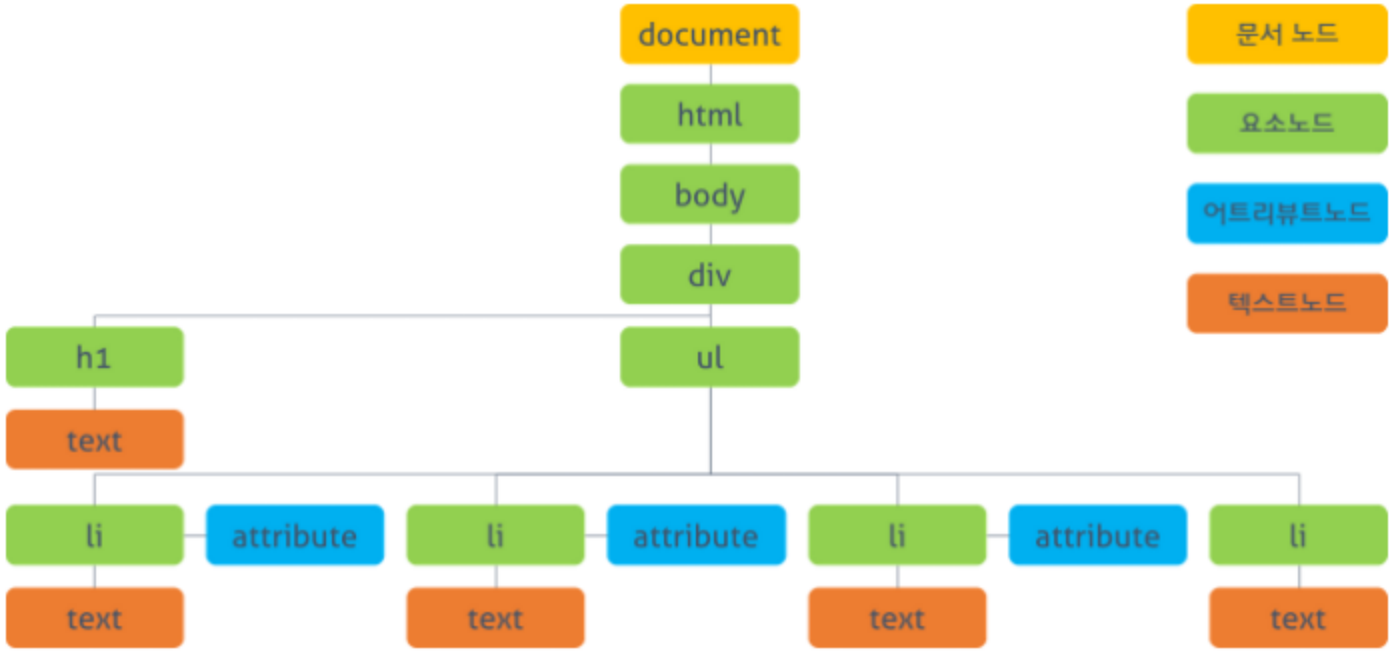
저번에 우리는

DOM

DOM (Document Object Model) 문서객체모델

가장 꼭대기(최상위)에 있는 노드를 루트 노드(root node)

document는 노드가 아니라 객체이므로 여기서는 html을 루트 노드라 부름



콘텐츠 조작하기

속성	설명
textContent	요소 노드의 모든 텍스트에 접근합니다.
innerText	요소 노드의 텍스트 중 웹 브라우저에 표시되는 텍스트에만 접근합니다.
innerHTML	요소 노드의 텍스트 중 HTML 태그를 포함한 텍스트에만 접근합니다.

```
<p id="title">Hello, <span style="display:none;">Javascript!</span></p>
```

```
document.getElementById("title").textContent; // Hello, Javascript!
document.getElementById("title").innerText;   // Hello,
document.getElementById("title").innerHTML;   // Hello, <span style="display: none;">Javascript!</span>
```

노드 추가/삭제

노드 추가

구분	메서드	설명
노드 생성	createElement()	요소 노드를 생성합니다.
	createTextNode()	텍스트 노드를 생성합니다.
	createAttribute()	속성 노드를 생성합니다.
노드 연결	<기준 노드>.appendChild(<자식 노드>)	기준 노드에 자식 노드를 연결합니다.
	<기준 노드>.setAttributeNode(<속성 노드>)	기준 노드에 속성 노드를 연결합니다.

```
<!DOCTYPE html>
<html>
<head>
    <title>Create Node</title>
</head>
<body>
    <script></script>
</body>
</html>
```

이벤트 리스너

주요 이벤트데이터 속성 조작하기

웹 브라우저에서 사용자와의 상호작용으로 발생하는 이벤트는 200여 가지가 넘음

구분	이벤트	설명
마우스 이벤트	onclick	마우스로 클릭하면 발생합니다.
	ondblclick	마우스로 빠르게 두 번 클릭하면 발생합니다.
	onmouseover	마우스 포인터를 올리면 발생합니다.
	onmouseout	마우스 포인터가 빠져나가면 발생합니다.
	onmousemove	마우스 포인터가 움직이면 발생합니다.
	onwheel	마우스 휠(wheel)을 움직이면 발생합니다.
키보드 이벤트	onkeypress	키보드 버튼을 누르고 있는 동안 발생합니다.
	onkeydown	키보드 버튼을 누른 순간 발생합니다.
	onkeyup	키보드 버튼을 눌렀다가 뗀 순간 발생합니다.
포커스 이벤트	onfocus	요소에 포커스가 되면 발생합니다.
	onblur	요소가 포커스를 잃으면 발생합니다.
폼 이벤트	onsubmit	폼이 전송될 때 발생합니다.

실습 환경

실습환경

OS : Window / Mac

브라우저 : Chrome

에디터 : VS Code https://code.visualstudio.com/

VS Code 익스텐션 : ESLint, Prettier, HTML CSS Support, HTML to CSS autocompletion, Auto Rename Tag, Auto Close Tag, htmltagwrap

Git : git bash, github 가입

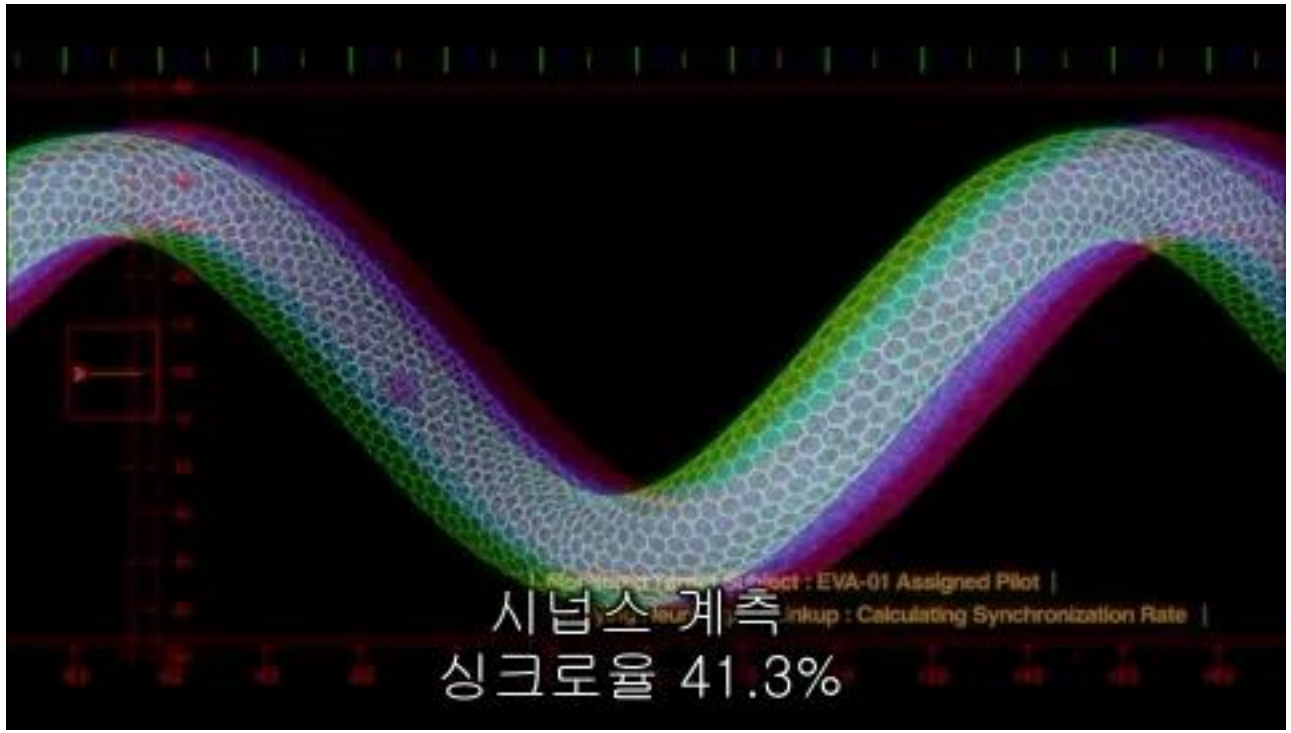
동기와 비동기

동기화...

동기와 비동기

동기화...

에반게리온?

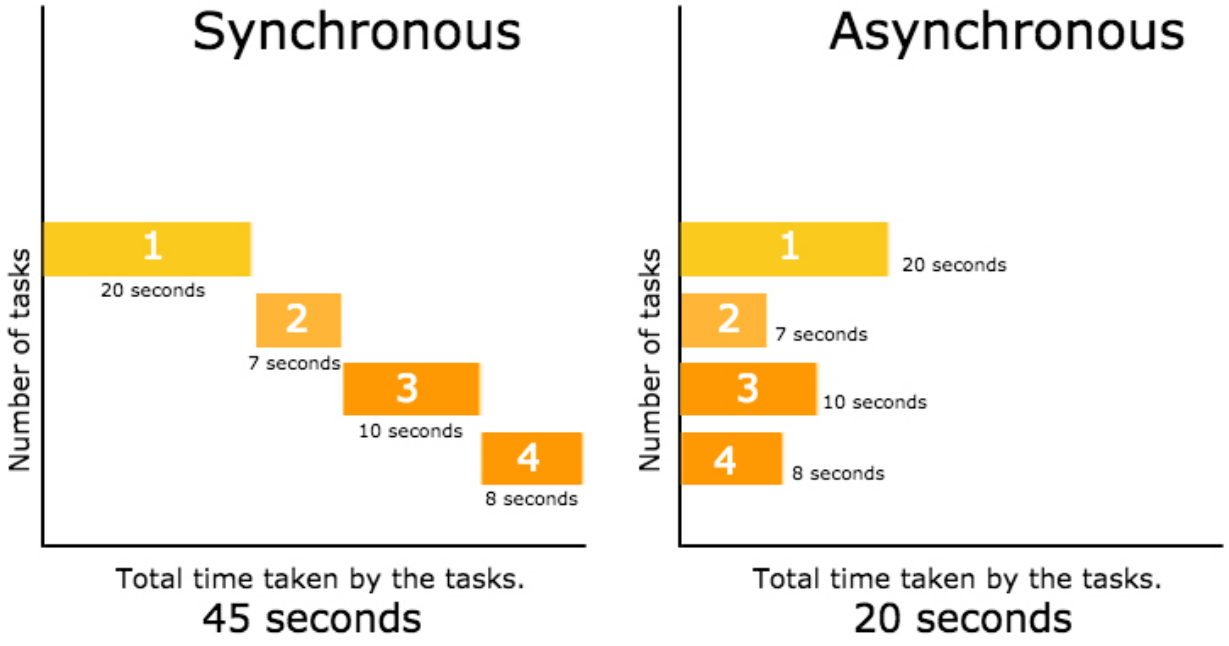


동기와 비동기

동기와 비동기 개념 이해

동기 : 동기는 순차적으로 일이 진행 되는 것

비동기 : 작업이 완료되기를 기다리지 않고 다음 작업을 실행



동기와 비동기

동기 Synchronous

코드가 순차적으로 실행되며, 한 작업이 완료될 때까지 다음 작업 대기

```
console.log("시작");
function longRunningTask() {
    let sum = 0;
    for (let i = 0; i < 100; i++) {
        sum += i;
    }
    return sum;
}
console.log("계산 시작");
console.log(longRunningTask());
console.log("끝");
```

동기와 비동기

비동기 Asynchronous

작업이 완료되기를 기다리지 않고 다음 작업을 실행

- 사용자 경험 개선: UI가 멈추지 않고 반응 가능.
- 성능 최적화: 시간이 오래 걸리는 작업을 백그라운드에서 처리.
- 예: 웹에서 이미지 로딩, 데이터 가져오기.

```
console.log("이미지 로딩 시작");
setTimeout(() => console.log("이미지 로딩 완료"), 2000);
console.log("다른 작업 수행");
```

동기와 비동기

동기와 비동기 개념 이해

자바스크립트는 단일 스레드 기반: 한 번에 하나의 작업만 처리.

동기 처리만 사용하면 시간이 오래 걸리는 작업(예: API 호출, 파일 로드)이 프로그램을 멈추게 함.

비동기 동작의 필요성:

- 사용자 경험(UX) 개선: 페이지 로딩 중 사용자 입력 가능.
- 성능 최적화: 작업 병렬 처리로 효율성 증가.

동기와 비동기

웹 개발에서 동기와 비동기

동기(Sync) 통신은 웹페이지를 새로고침하면서 데이터를 불러오는 방식
동기(Sync) 통신의 가장 대표적인 예시가 폼(Form) 제출
폼(Form)은 웹페이지에서 사용자의 입력을 받아 서버에 전송하는 역할
클라이언트에서 폼(Form)을 서버에 제출하면 서버로부터 새로운 웹페이지를 받아서 화면에 표시

웹페이지를 새로고침하지 않고도 데이터를 불러오는 방식
네이버 블로그나 카페에서 댓글을 작성하고 작성 버튼을 누르면 페이지 전체가 재로드 되지 않고 댓글 영역 부분만 업데이트 되어 댓글이 적용됨

동기와 비동기

웹 개발에서 동기와 비동기

동기(Sync)로 페이지 이동 표현

/main

/blog

/calc

비동기(Async)로 데이터 조회, 가공,

/blog

Blog 글 조회

Blog 글 조회수 상승

Blog 글 생성요청

Server

비동기처리

비동기처리

Callback 함수

Promise

Async/await

콜백함수

다른 함수의 인자로 전달되어 특정 작업이 완료된 후 호출되는 함수.
비동기 작업의 결과를 처리하거나, 특정 이벤트 발생 시 실행.
자바스크립트에서 비동기 처리의 기본 도구.
예: 버튼 클릭 시 실행되는 이벤트 핸들러, 타이머 완료 후 호출되는 함수.

```
function sayHello(name, callback) {
    console.log(`안녕, ${name}!`);
    callback();
}
sayHello("학생", () => console.log("콜백 실행!"));
```

비동기처리

콜백함수

DOM 이벤트 처리: 버튼 클릭, 마우스 이동.

비동기 요청: 서버에서 데이터 가져오기.

타이머: setTimeout, setInterval.

```
<button id="btn">클릭!</button>
<script>
    document.getElementById("btn").addEventListener("click", () => {
        console.log("버튼 클릭됨!");
    });
</script>
```

콜백지옥

비동기 작업을 처리하기 위해 콜백 함수를 깊게 중첩하면 코드가 복잡해지는 현상.

- 들여쓰기 증가로 가독성 저하.
- 에러 처리 복잡, 디버깅 어려움.
- 복잡한 로직: 순차적 작업이 많아질수록 관리 어려움.

```
getUser(user => {
    getPosts(user, posts => {
        getComments(posts, comments => {
            console.log(comments);
        });
    });
});
```

Promise

비동기 작업의 성공/실패를 관리하는 객체.

상태:

- Pending: 작업 진행 중.
- Fulfilled: 작업 성공.
- Rejected: 작업 실패.

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("작업 성공!"), 2000);
});
promise.then(result => console.log(result));
```

비동기처리

Promise chaining

.then()으로 성공 결과 처리, .catch()로 에러 처리.
체이닝으로 순차적 비동기 작업 관리.

```
new Promise((resolve) => {
    setTimeout(() => resolve(1), 1000);
})
    .then(result => {
        console.log(result);
        return result + 1;
    })
    .then(result => console.log(result))
    .catch(error => console.log("에러:", error));
```

Promise.all로 병렬 처리

Promise.all: 여러 Promise를 병렬로 실행, 모두 완료될 때까지 대기.
모든 Promise가 성공하면 결과 배열 반환, 하나라도 실패하면 즉시 에러

```
const p1 = new Promise(resolve => setTimeout(() => resolve("작업 1"), 1000));
const p2 = new Promise(resolve => setTimeout(() => resolve("작업 2"), 2000));
Promise.all([p1, p2]).then(results => console.log(results));
```

Promise.all로 병렬 처리

Promise.all: 여러 Promise를 병렬로 실행, 모두 완료될 때까지 대기.

모든 Promise가 성공하면 결과 배열 반환, 하나라도 실패하면 즉시 에러

```
const p1 = new Promise(resolve => setTimeout(() => resolve("작업 1"), 1000));
const p2 = new Promise(resolve => setTimeout(() => resolve("작업 2"), 2000));
Promise.all([p1, p2]).then(results => console.log(results));
```

Promise.all로 병렬 처리

```
const allTest = () => {
    const p1 = new Promise(resolve => setTimeout(() => resolve("작업 1"), 1000));
    const p2 = new Promise(resolve => setTimeout(() => resolve("작업 2"), 2000));
    Promise.all([p1, p2]).then(results => console.log(results));
};
allTest();
```

Promise.race와 Promise.any

Promise.race: 가장 먼저 완료된 Promise의 결과 반환(성공/실패 모두).

Promise.any: 가장 먼저 성공한 Promise의 결과 반환, 모두 실패 시 에러.

```
const p1 = new Promise(resolve => setTimeout(() => resolve("느린 성공"), 2000));
const p2 = new Promise((_, reject) => setTimeout(() => reject("빠른 실패"), 1000));
Promise.race([p1, p2]).then(results => console.log(results)).catch(error => console.log(error));
Promise.any([p1, p2]).then(result => console.log(result)).catch(error => console.log(error));
```

비동기처리


```
const raceTest = () => {
    const p1 = new Promise(resolve => setTimeout(() => resolve("느린 성공"), 2000));
    const p2 = new Promise((_, reject) => setTimeout(() => reject(new Error("빠른 실패")), 1000));
    Promise.race([p1, p2])
        .then(result => console.log("Race resolved:", result))
        .catch(error => console.log("Race rejected:", error.message));
};
raceTest();
const anyTest = () => {
    const p1 = new Promise(resolve => setTimeout(() => resolve("느린 성공"), 2000));
    const p2 = new Promise((_, reject) => setTimeout(() => reject(new Error("빠른 실패")), 1000));
    Promise.any([p1, p2])
        .then(result => console.log("Any resolved:", result))
        .catch(error => console.log("Any rejected:", error.message));
};
anyTest();
```

비동기처리

Async / Await

async: 함수가 Promise를 반환하도록 설정.

await: Promise가 완료될 때까지 대기, 동기처럼 코드 작성 가능.

Promise 대비 가독성과 유지보수성 향상.

```
async function fetchData() {
    let promise = new Promise(resolve => setTimeout(() => resolve("데이터 가져옴!"), 2000));
    let result = await promise;
    console.log(result);
}
fetchData();
```

Async / Await 에러 처리

try/catch로 async/await에서 에러 처리.

Promise의 .catch()보다 직관적.

```
async function fetchWithError() {
    try {
        let response = await new Promise((_, reject) => {
            setTimeout(() => reject("에러 발생!"), 1000);
        });
    } catch (error) {
        console.log("에러:", error);
    }
}
fetchWithError();
```

Fetch Api

fetch: HTTP 요청을 보내고 Promise 반환.

기본 문법: fetch(url).then(response => response.json()).

실제 API 호출로 데이터 가져오기.

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(response => response.json())
    .then(data => console.log(data.title))
    .catch(error => console.log("에러:", error));
```

Async / Await 으로 Fetch Api 간소화

https://www.notion.so/fetch-api-229caf5650aa80bc80bdf20b790f5b04

```
let response = await fetch("https://jsonplaceholder.typicode.com/posts/2");
let data = await response.json();
console.log(data.title);
```

오늘 우리는

동기와 비동기

웹 개발에서 동기와 비동기

동기(Sync)로 페이지 이동 표현

/main

/blog

/calc

비동기(Async)로 데이터 조회, 가공,

/blog

Blog 글 조회

Blog 글 조회수 상승

Blog 글 생성요청

Server

콜백함수

다른 함수의 인자로 전달되어 특정 작업이 완료된 후 호출되는 함수.
비동기 작업의 결과를 처리하거나, 특정 이벤트 발생 시 실행.
자바스크립트에서 비동기 처리의 기본 도구.
예: 버튼 클릭 시 실행되는 이벤트 핸들러, 타이머 완료 후 호출되는 함수.

```
function sayHello(name, callback) {
    console.log(`안녕, ${name}!`);
    callback();
}
sayHello("학생", () => console.log("콜백 실행!"));
```

Promise

비동기 작업의 성공/실패를 관리하는 객체.

상태:

- Pending: 작업 진행 중.
- Fulfilled: 작업 성공.
- Rejected: 작업 실패.

```
const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("작업 성공!"), 2000);
});
promise.then(result => console.log(result));
```

비동기처리

Async / Await

async: 함수가 Promise를 반환하도록 설정.

await: Promise가 완료될 때까지 대기, 동기처럼 코드 작성 가능.

Promise 대비 가독성과 유지보수성 향상.

```
async function fetchData() {
    let promise = new Promise(resolve => setTimeout(() => resolve("데이터 가져옴!"), 2000));
    let result = await promise;
    console.log(result);
}
fetchData();
```

Async / Await 으로 Fetch Api 간소화

https://www.notion.so/fetch-api-229caf5650aa80bc80bdf20b790f5b04

```
let response = await fetch("https://jsonplaceholder.typicode.com/posts/2");
let data = await response.json();
console.log(data.title);
```

감사합니다