

Javascript 기초

ES6+와 JSON, 예외처리

김태민

저번에 우리는

웹 개발에서 동기와 비동기

동기(Sync)로 페이지 이동



비동기(Async)로 데이터 조회, 가공, 표현



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));
```

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);
```

실습 환경

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

ES6+와 JSON

ECMAScript

ECMAScript는 자바스크립트의 표준 사양

ES6(2015)부터 화살표 함수, let/const, 모듈 등 현대적 기능 추가

ES6+는 ES2015 이후 버전, 웹 개발 필수

```
const greet = (name) => `안녕, ${name}!`;
console.log(greet("철수")); // 출력: 안녕, 철수!
```

ECMAScript 주요 특징

let/const: 블록 스코프 변수

화살표 함수: 간결한 함수 표현

구조 분해 할당, 스프레드 연산자: 데이터 조작 간소화

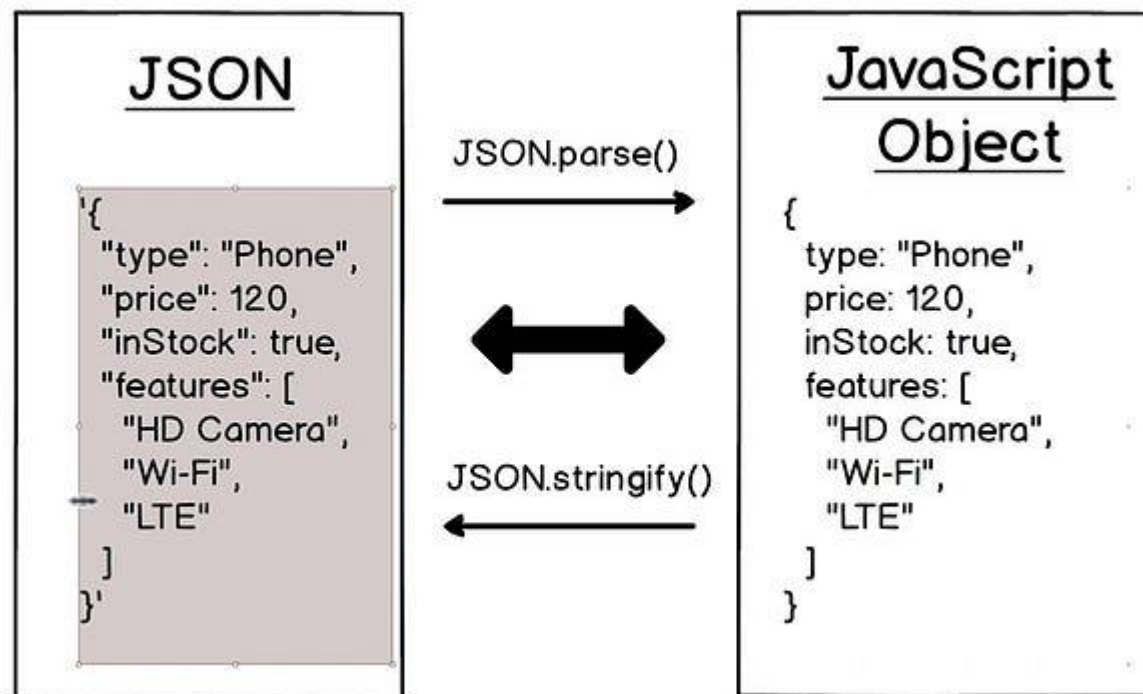
async/await: 비동기 처리 개선

```
const user = { name: "영희", age: 25 };  
const { name, age } = user;  
console.log(name, age); // 출력: 영희 25
```

JSON과 객체리터럴

JSON: 문자열, {}(객체) 또는 [](배열), 엄격한 구문

객체 리터럴: 자바스크립트 코드, {}로 객체 생성, 함수 포함 가능



JSON과 객체리터럴

JSON (JavaScript Object Notation):

데이터 교환을 위한 경량 포맷으로, 문자열 기반의 독립적인 데이터 구조.

1990년대 말 정의, ES5(2009)에서 JSON.parse와 JSON.stringify 표준화.

자바스크립트뿐 아니라 다른 언어(Python, Java 등)에서도 사용 가능.

주로 API 응답, 데이터 저장/전송에 사용.

```
{  
  "name": "민수",  
  "age": 30,  
  "hobbies": ["코딩", "운동"]  
}
```

```
[  
  {  
    "id": 1,  
    "name": "영희"  
  },  
  {  
    "id": 2,  
    "name": "민수"  
  }  
]
```

JSON과 객체리터럴

Object Literal:

자바스크립트 코드로 작성되며, 키에 큰따옴표를 생략 가능(유효한 식별자인 경우).
값으로 함수, undefined, Symbol 등 자바스크립트의 모든 데이터 타입을 포함 가능

```
const obj = {  
  name: "철수",  
  age: 25,  
  greet: () => "안녕!"  
};
```

JSON과 객체리터럴

JSON.parse()

JSON 문자열을 객체로 변환

JSON.parse로 서버 데이터 처리

ES6+ 구조 분해로 데이터 추출 간편

```
const jsonString = '{"name": "지민", "age": 22}';  
const { name, age } = JSON.parse(jsonString);  
console.log(name, age); // 출력: 지민 22
```

```
const jsonString = '[{"id": 1, "name": "혜진"}, {"id": 2, "name": "민수"}]';  
const users = JSON.parse(jsonString);  
const names = users.map(({ name }) => name);  
console.log(names); // 출력: ["혜진", "민수"]
```


JSON과 객체리터럴

JSON.stringify()

객체를 JSON 문자열로 변환

JSON.stringify로 데이터 직렬화

ES6+로 객체 생성 간결화

```
const obj = { name: "유진", age: 27 };  
const arr = [{ id: 1, name: "지민" }, { id: 2, name: "수진" }];  
console.log(JSON.stringify(obj, null, 2));  
console.log(JSON.stringify(arr, null, 2));
```

JSON과 객체리터럴

JSON 데이터 조작

Map을 사용하여 JSON 배열 처리 가능

```
const jsonString = '[{"id": 1, "name": "혜진"}, {"id": 2, "name": "민수"}]';  
const users = JSON.parse(jsonString);  
const names = users.map(({ name }) => name);  
console.log(names); // 출력: ["혜진", "민수"]
```

예외처리

예외처리의 중요성

애플리케이션 안정성 보장

사용자 경험 개선

디버깅, 보안 등 사용자 신뢰에 영향



오류 Error

요청하신 페이지를 처리 중에 오류가 발생했습니다. 서비스 이용에 불편을 드려 죄송합니다.
입력하신 주소가 정확한지 확인 후 다시 시도해 주시기 바랍니다.

We have encountered a system error while processing your request.
We apologize for the inconvenience. Please check URL and try again.

Try-Catch

```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

```
}
```

```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

```
}
```

```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

```
} finally {  
    // 항상 실행  
}
```

Try-Catch 사용자입력 검증

사용자가 입력한 데이터가 예상과 다를 때 발생하는 오류 처리.

예: 숫자가 아닌 값을 입력하거나 필수 필드가 누락된 경우.

```
try {  
  const input = "not-a-number";  
  const number = Number(input);  
  if (isNaN(number)) throw new Error("숫자를 입력하세요.");  
  console.log("입력값:", number);  
} catch (error) {  
  console.log("에러:", error.message);  
}
```

Try-Catch 네트워크 요청 처리

파일 읽기/쓰기, API 호출 등 외부 리소스와의 상호작용에서 발생하는 오류 처리.

예: 네트워크 연결 실패, 잘못된 API 응답.

```
async function fetchUser() {  
  try {  
    const response = await fetch("https://invalid-url");  
    if (!response.ok) throw new Error("네트워크 오류 발생");  
    const data = await response.json();  
    console.log(data);  
  } catch (error) {  
    console.log("API 에러:", error.message);  
  }  
}  
  
fetchUser();
```

Try-Catch 데이터 파싱

JSON 파싱, XML 파싱 등 데이터 변환 중 오류 처리.

예: 잘못된 JSON 형식으로 인한 `SyntaxError`.

```
try {  
  const json = '{"name": "철수"}'; // 잘못된 JSON  
  const data = JSON.parse(json);  
  console.log(data);  
} catch (error) {  
  console.log("파싱 에러:", error.message);  
}
```


Try-Catch 리소스 접근

DOM 요소, 파일 시스템, 데이터베이스 등 접근 시 발생하는 오류 처리.

예: 존재하지 않는 DOM 요소에 접근.

```
try {  
    const element = document.getElementById("nonexistent");  
    if (!element) throw new Error("요소가 존재하지 않습니다.");  
    element.textContent = "업데이트됨";  
} catch (error) {  
    console.log("DOM 에러:", error.message);  
}
```

Try-Catch 비동기 작업에서의 예외처리

ES6+의 async/await를 사용한 비동기 작업에서 오류 처리.

예: Promise 거부(reject) 처리.

```
async function processData() {  
  try {  
    const promise = Promise.reject("작업 실패");  
    await promise;  
  } catch (error) {  
    console.log("비동기 에러:", error);  
  }  
}  
processData();
```

Try-Catch 외부 라이브러리 사용

서드파티 라이브러리나 모듈 호출 시 예상치 못한 오류 처리.

예: 라이브러리 메서드가 예외를 던지는 경우.

```
try {  
    const result = someLibrary.invalidMethod(); // 존재하지 않는 메서드  
    console.log(result);  
} catch (error) {  
    console.log("라이브러리 에러:", error.message);  
}
```

Try-Catch 비즈니스 로직 검증

애플리케이션의 특정 조건을 만족하지 않을 때 커스텀 예외 처리.

예: 주문 금액이 최소값 미만인 경우.

```
try {  
    const orderAmount = 500;  
    if (orderAmount < 1000) throw new Error("최소 주문 금액은 1000원입니다.");  
    console.log("주문 처리 완료");  
} catch (error) {  
    console.log("주문 에러:", error.message);  
}
```

계산기 제작

계산기 서비스 목표

숫자판(0~9)과 연산 키(+, -, ×, ÷)로 사칙연산 수행

JSON 배열([])로 계산 기록 저장

제공한 코드 기반 구체적인 로직 직접 생성



계산기 서비스 제작

Github Repository 생성

Repository Name : oz-calculator

공개여부 : public

계산기 서비스 프로젝트 생성

프로젝트명 : oz-calculator

GitHub 연결

```
git remote add origin {git-repository-url}
```

```
git remote -v
```

```
git status
```

```
git add .
```

```
git commit -m "init repository"
```

```
git push origin main
```


계산기 서비스 프로젝트 제공코드

제공 코드:

index.html: Bootstrap 5 기반 UI (숫자판, 연산 키, 디스플레이)

calculator.js: 숫자 입력, 연산자 설정, 초기화, 에러 출력 틀

학생 작성 범위:

계산 로직 (사칙연산)

예외처리 (숫자 유효성, 0으로 나누기, 연산자 유효성 등)

<https://github.com/GoodPineApple/oz-calculator>

오늘 우리는

ECMAScript

ECMAScript는 자바스크립트의 표준 사양

ES6(2015)부터 화살표 함수, let/const, 모듈 등 현대적 기능 추가

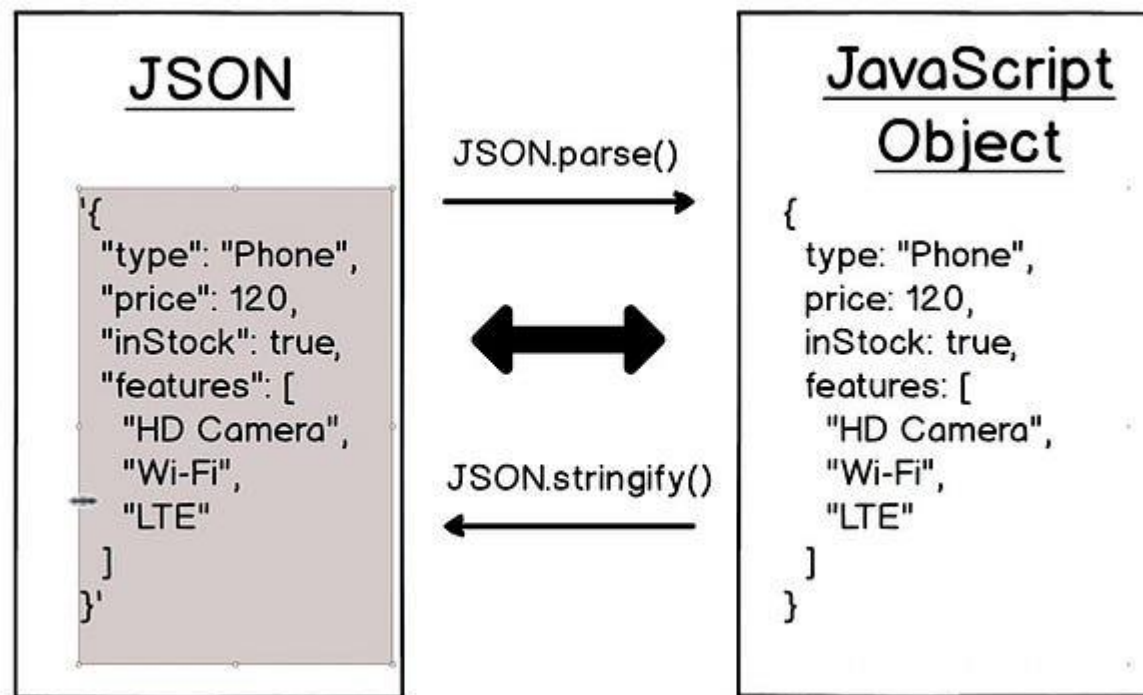
ES6+는 ES2015 이후 버전, 웹 개발 필수

```
const greet = (name) => `안녕, ${name}!`;
console.log(greet("철수")); // 출력: 안녕, 철수!
```

JSON과 객체리터럴

JSON: 문자열, {}(객체) 또는 [](배열), 엄격한 구문

객체 리터럴: 자바스크립트 코드, {}로 객체 생성, 함수 포함 가능



Try-Catch

```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

```
}
```



```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

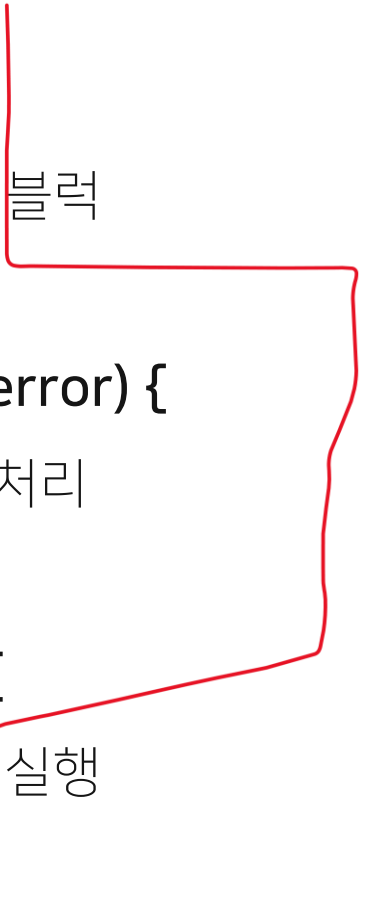
```
}
```



```
try {  
    // 코드 블록
```

```
} catch (error) {  
    // 예외처리
```

```
} finally {  
    // 항상 실행  
}
```



감사합니다