**HTML의 form과 action의 개념**

     - method

     - GET / POST

     - parameter (name, value)

**1.form = 데이터를 전달하는 '구조'**

* form은 사용자로부터 **입력받은 데이터**를 **서버**로 전달하는 HTML 요소.
* method와 action 속성은 이 데이터를 **어떻게, 어디로** 보낼지를 정함
  + **method**: 데이터를 "어떻게" 보낼지 (GET, POST 등).
  + **action**: 데이터를 "어디로" 보낼지 (서버 주소).

**2. action**

* **action**은 데이터를 보낼 **목적지**를 지정.  
  예: action="/submit"
  + 사용자가 데이터를 입력하고 제출하면, /submit URL로 데이터를 보낸다.

**예시: GET과 POST 차이**

**<form action="/submit" method="GET">**

<**input** type="text" name="name" placeholder="Enter your name">

<button type="submit">Send</button>

</form>

* **결과**:
  + 사용자가 이름을 "John"으로 입력하고 제출하면 브라우저가 **/submit?name=John**으로 요청을 보낸다.

**<form action="/submit" method="POST">**

<**input** type="text" name="name" placeholder="Enter your name">

<button type="submit">Send</button>

</form>

* **결과**:
  + 동일한 데이터를 보내도 URL**은 /submit으로 유지되고**, 데이터는 요청의 *본문(body)에 포함*되어 서버로 전송된다.

**3. GET, POST in method of form**

* **GET**
  + 데이터를 **URL의 쿼리스트링**(예: ?name=John)으로 전달.

\* 쿼리스트링은 URL의 **"?" 이후에 나타나는 데이터**로, **키-값 쌍(key-value pairs)**으로 이루어져 있다.

* + **URL로 표시되기 때문에** 데이터를 바로 확인할 수 있음.
  + 예: 검색창, 간단한 조회 작업.
* **POST**
  + 데이터를 **HTTP 요청의 본문(body)**에 포함해서 전달.
  + **URL에** 표시되지 **않**으므로 비밀번호 같은 *민감한 정보*를 안전하게 보낼 때 사용.
  + 예: 로그인, 회원가입, 파일 업로드.

**한 줄 정리**

action이 데이터를 전달할 URL을 지정하고, method가 데이터 전달 방식을 결정한다.  
form은 **action**(어디로)과 **method**(어떻게)로 데이터를 전달하는 도구이며, method에는 주로 **GET과 POST**가 있다!

**요약**

* **form**: 데이터를 사용자로부터 입력받아 전송하는 HTML 구조.
* **action**: 데이터를 보낼 목적지(URL).
* **method**: 데이터를 어떻게 보낼지(전송 방식: GET 또는 POST).

form은 action과 method로 설계된 구조.

**parameter (name, value)**

HTML에서 **parameter (name, value)**는 태그에 정보를 추가하는 역할.  
쉽게 말해, 태그가 "어떻게 동작해야 하는지" 알려줌.  
\*태그의 형식: **<와 >**로 감싸여 있다.

**기본 형식**

name="value"

* **name**: 설정의 이름
* **value**: 설정의 값

**예시**

**1. 링크**

<a href="https://example.com">이동하기</a>

* href (name): 링크의 주소를 알려줌
* https://example.com (value): 이동할 URL

**2. 이미지**

<img src="image.jpg" alt="예쁜 그림">

* src (name): 이미지의 파일 경로
* image.jpg (value): 표시할 이미지 파일
* alt (name): 이미지가 안 보일 때 대신 나올 텍스트

**3. 입력폼**

<form action="/submit" method="GET">

<input type="text" name="username" value="학생">

<button type="submit">보내기</button>

</form>

* name (name): 서버로 보낼 데이터의 이름 (username)
* value (value): 기본 값 (학생)

**요약**

* **name**: 설정의 이름 (뭘 설정할지 알려줌)
* **value**: 설정의 값 (어떤 값을 사용할지 알려줌)
* 예: href="https://example.com"은 링크가 어디로 가야 하는지 알려주는 역할!

**참고**

HTML에서는 **name="value"** 형식을 사용하지만, parameter (name, value)라고 표현한 이유는 **프로그래밍 전반에서 "파라미터"를 설명할 때 흔히 사용하는 표현 방식** 때문.

**Javascript**

  - DOM Manipulation

    - EventListener (이 부분은 onClick과 비교)

    - Event loop

    - getelementByID

    - getelementsByClassName

    - queryselector   vs  queryselectorAll

    - Node의 속성

**DOM Manipulation (돔 조작)**

**DOM**은 "Document Object Model"의 줄임말로, 웹 페이지를 자바스크립트로 조작할 수 있게 해주는 구조이다.  
웹 페이지의 HTML 태그들이 모두 자바스크립트 객체로 변환되어 있기 때문에, 이걸 이용해서 내용을 바꾸거나, 새로운 요소를 추가하거나, 삭제할 수 있다.

**EventListener와 onClick**

**EventListener**는 웹 페이지에서 일어나는 특정 이벤트(클릭, 키보드 입력, 마우스 움직임 등)를 감지하고, 그에 따라 동작하도록 코드를 작성할 때 사용.

* onClick은 버튼을 클릭했을 때 특정 동작을 하도록 설정하는 방식인데, 코드에 직접 추가하거나 HTML 속성으로 추가한다.
* **차이점**:
  + EventListener는 한 요소에 여러 개의 동작을 추가할 수 있다.
  + onClick은 하나만 추가 가능하고, 나중에 덮어쓰일 수도 있다.

// EventListener

document.getElementById("myButton").**addEventListener("click", function () {**

**alert("버튼 클릭!");**

});

mybutton.addEventListener("click", () => console.log("두 번째 핸들러"));

// onClick

const button = document.querySelector("button");

button.**onclick = () => console.log("첫 번째 핸들러");**

// 기존 핸들러를 덮어씀

button.onclick = () => console.log("두 번째 핸들러");

// onClick in HMTL

<button onclick="alert('버튼 클릭!')">클릭</button>

**Event Loop (이벤트 루프)**

자바스크립트는 한 번에 하나의 작업만 처리하는 "싱글 스레드" 언어.  
하지만 이벤트(클릭, 타이머, 데이터 요청 등)가 많으면?

**Event Loop**는 이 작업들을 순서대로 처리해주는 "대기 줄" 같은 역할.  
1. **Call Stack**: 지금 실행 중인 코드가 쌓이는 곳.  
2. **Task Queue**: 이벤트나 비동기 작업이 끝난 후 실행 대기 중인 코드들이 모이는 곳.  
3. Event Loop가 "Call Stack"이 비었는지 확인하고, 비어 있으면 "Task Queue"에서 코드를 꺼내 실행.

**getElementById**

**getElementById**는 **id** 속성을 가진 HTML 요소를 선택할 때 사용해. id는 각 요소마다 고유해야 하기 때문에 특정 요소를 빠르게 찾음.

const title = document.getElementById("title");

title.style.color = "red"; // id가 "title"인 요소의 텍스트 색을 빨간색으로 변경

**getElementsByClassName**

**getElementsByClassName**은 **class** 속성을 가진 HTML 요소들을 선택.  
같은 클래스를 가진 요소가 여러 개 있을 수 있으니, 배열처럼 사용.  
(왜 배열처럼 사용해야 하냐면  
"item"이 여러 개니까, **하나씩** 다루려면 **순서**를 써야 해.  
이런 경우, for 같은 반복문으로 하나씩 꺼내서 처리!)

const items = document.getElementsByClassName("item");

for (let item of items) {

item.style.color = "blue"; // 모든 "item" 클래스를 가진 요소의 텍스트 색을 파란색으로 변경

}

**querySelector vs querySelectorAll**

* **querySelector**: 선택한 CSS 선택자에 해당하는 **첫 번째 요소**를 가져옴.
* **querySelectorAll**: 선택한 CSS 선택자에 해당하는 **모든 요소**를 가져와서 배열처럼 다룸.

const firstItem = document.querySelector(".item"); // 첫 번째 "item" 클래스 요소

const allItems = document.querySelectorAll(".item"); // 모든 "item" 클래스 요소를 노드리스트로 담음

여기서 말하는 **CSS 선택자** = HTML 요소를 **찾는 방법**.  
= CSS에서 스타일을 적용할 때 *어떤 요소에 적용할지 정하는* **규칙** = **"선택자"**.  
자바스크립트의 querySelector와 querySelectorAll에서도 같은 방식으로 요소를 선택해.

**CSS 선택자의 종류는?**

1. **태그 선택자**: 태그 이름으로 선택
   * 예: div, p, h1 등
   * 사용: 페이지 안의 모든 해당 태그를 선택

document.querySelector("p"); // 첫 번째 <p> 태그

1. **클래스 선택자**: .(점)으로 클래스 이름을 선택
   * 예: .my-class는 class="my-class"를 가진 요소를 선택

document.querySelector(".my-class"); // 첫 번째 "my-class" 클래스

1. **ID 선택자**: #으로 ID 이름을 선택
   * 예: #my-id는 id="my-id"를 가진 요소를 선택

document.querySelector("#my-id"); // id="my-id"인 요소

1. **조합 선택자**: 여러 선택자를 조합해서 특정 요소를 선택
   * 예: div.my-class는 <div> 태그 중 class="my-class"를 가진 것만 선택

document.querySelector("div.my-class"); // div 태그 중 class="my-class"인 첫 번째 요소

1. **자식/후손 선택자**: 계층 구조에 따라 선택
   * >: 바로 아래 자식
   * 공백: 모든 후손
2. document.querySelector("div > p"); // div의 "직접 자식"인 <p> 태그

document.querySelector("div p"); // div 안의 모든 <p> 태그

**결론**  
CSS 선택자 = "어떤 요소를 선택할지"를 정하는 규칙,  
querySelector와 querySelectorAll은 이 규칙을 이용해 원하는 요소를 찾아오는 도구!

**Node의 속성**

Node는 DOM의 기본 단위인데, HTML 요소(태그), 속성, 텍스트 등을 포함.

주요 속성들

* nodeName: 노드의 이름(예: DIV, SPAN)
* nodeType: 노드가 어떤 종류인지 나타내는 숫자(예: 1은 요소, 3은 텍스트)
* childNodes: 노드의 자식 노드 목록
* parentNode: 노드의 부모 노드

const node = document.getElementById("title");

console.log(node.nodeName); // "H1"

console.log(node.parentNode); // 부모 요소

**여기서 노드란?**

HTML 문서를 **"조각조각 나눈 기본 단위"**!

웹 페이지에서 모든 것이 노드!

HTML 태그도 노드  
텍스트도 노드  
주석도 노드

**노드의 종류**

요소 노드: HTML 태그 (예: < div >, < p >)  
텍스트 노드: 태그 안에 들어있는 글자 (예: "안녕하세요")  
주석 노드: HTML 주석 (예: < !-- 주석 -- >)

**요약:**

* DOM은 HTML 문서를 자바스크립트로 조작할 수 있는 도구.
* 이벤트를 처리할 때는 EventListener를 쓰는 게 더 유연하고 안전.
* 요소 선택은 getElementById, querySelector 등으로.
* 이벤트 루프는 자바스크립트가 효율적으로 작업을 처리하게 도와주는 시스템.

**Javascript 문법**

    - [var와 let의 차이](https://cafe.naver.com/codingstudyplace/77)

    - const

    - ES5의 Array 관련 메소드 (최소 4개 이상)

    - rest parameter

    - Arrow function

    - [1급 객체](https://cafe.naver.com/codingstudyplace/10)

- Object의 개념

**1. var와 let의 차이**

**var**

* **옛날 방식**: 자바스크립트 초창기부터 사용되던 방식.
* **특징 1: 블록 스코프를 무시**. (이유: 함수 스코프라서)  
  블록 {} 안에서 선언하더라도, 블록 밖에서도 사용가능.

{

var x = 10; // 블록 안에서 선언

}

console.log(x); // 10 (블록 밖에서도 접근 가능)

* **특징 2: 같은 이름의 변수 선언 가능**  
  같은 이름으로 변수를 여러 번 선언해도 에러 없다.

var a = 5;

var a = 10; // 다시 선언 가능

console.log(a); // 10

**let**

* **요즘 방식**: var의 단점을 개선한 새로운 방식.
* **특징 1: 블록 스코프 준수**  
  let으로 선언한 변수는 블록 {} 안에서만 접근 가능.

{

let y = 20; // 블록 안에서 선언

}

// console.log(y); // 에러 발생 (블록 밖에서는 접근 불가)

* **특징 2: 같은 이름의 변수 선언 불가**  
  이미 선언된 이름의 변수를 **다시 선언**하려고 하면 에러.

let b = 5;

let b = 10; // 에러 발생 (같은 이름 선언 불가)

* **특징 3: 선언 전에 사용할 수 없음**  
  var와 달리, let으로 선언한 변수는 선언하기 전에 접근하면 에러가 발생.  
  (이걸 "Temporal Dead Zone")

console.log(c); // 에러 발생

let c = 30;

**2. const**

* **상수 선언**: 변하지 않는 값을 저장할 때 사용.
* **특징 1: 값 변경 불가**  
  const로 선언한 변수는 값을 다시 할당할 수 없다.

const PI = 3.14;

PI = 3.15; // 에러 발생 (값 변경 불가)

* **특징 2: 객체와 배열의 내부는 변경 가능**  
  const는 변수 자체를 바꿀 수 없지만, **객체나 배열의 내부 값**은 바꿀 수 있다.

const person = { name: "Alice" };

**person.name = "Bob"; // 내부 값 변경 가능**

console.log(person.name); // Bob

const numbers = [1, 2, 3];

numbers.push(4); // 배열 수정 가능

console.log(numbers); // [1, 2, 3, 4]

* **특징 3: 선언과 동시에 초기화해야 함**  
  선언할 때 값을 지정하지 않으면 에러.

const age; // 에러 발생 (초기값 필요)

const age = 25; // 올바른 선언

**⭐요약 비교**

| **키워드** | **스코프** | **값 재할당** | **변수재선언** | **선언 전에 사용** |
| --- | --- | --- | --- | --- |
| var | 함수 스코프 | 가능 | 가능 | 가능 (undefined) |
| **let** | 블록 스코프 | 가능 | 불가능 | 불가능 (에러) |
| **const** | 블록 스코프 | 불가능 | 불가능 | 불가능 (에러) |

**언제 사용하면 좋을까?**

1. **let:** 일반적으로 변수가 변할 수 있는 경우.
2. **const:** 변하지 않는 값이나 객체, 배열을 사용할 때.
3. **var:** 옛날 코드 유지보수나 특수한 상황이 아니면 사용하지 않는 것이 좋다.

**참고**

**함수 스코프와 블록 스코프**

* **함수 스코프(Function Scope):** 변수가 **함수 안**에서만 접근 가능. 함수 밖에서는 보이지 않음. (var가 해당)

function test() {

var x = 10; // 함수 안에서 선언

}

console.log(x); // 에러 발생 (함수 밖에서 접근 불가)

* **블록 스코프(Block Scope):** 변수가 **블록 {} 안**에서만 접근 가능. 블록 밖에서는 보이지 않음. (let과 const가 해당)

{

let y = 20; // 블록 안에서 선언

}

console.log(y); // 에러 발생 (블록 밖에서 접근 불가)

**ES5에서 가장 자주 쓰이는 Array 메소드 5가지**

(\*ES5는 2009년에 나온 JavaScript 표준)

**1. forEach**

* 배열의 모든 요소를 순회하며 함수 실행.
* 주로 배열을 반복 처리할 때 사용.

const numbers = [1, 2, 3];

numbers.forEach((num) => console.log(num));

// 출력: 1, 2, 3

**2. map**

* 배열의 각 요소를 **변환**한 **새로운 배열 반환**.
* 원본 배열은 변경되지 않음.

const numbers = [1, 2, 3];

const doubled = numbers.map((num) => num \* 2);

console.log(doubled);

// 출력: [2, 4, 6]

**3. filter**

* 조건에 맞는 요소만 **선택**해 **새로운 배열 반환**.
* 원본 배열은 변경되지 않음.

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

const evens = numbers.filter((num) => num % 2 === 0);

console.log(evens);

// 출력: [2, 4]

**4. reduce (축약 메소드)**

* 배열의 모든 요소를 순회하며, 하나의 값으로 **누적**.
* 초기값(초기 누적값)을 지정할 수 있으며, 이를 통해 숫자 합계, 객체 병합, 문자열 연결 등 다양한 작업을 수행.

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

const sum = numbers.reduce((acc, num) => acc + num, 0); // 0은 초기값

console.log(sum);

// 출력: 10 (1 + 2 + 3 + 4)

* 예를 들어, 배열의 값을 곱하는 작업도.

const product = numbers.reduce((acc, num) => acc \* num, 1); // 초기값을 1로 설정

console.log(product);

// 출력: 24 (1 \* 2 \* 3 \* 4)

**5. some (조건 확인 메소드)**

* 배열에서 **조건에 맞는 요소가 하나라도 있으면 true 반환**.
* 모든 요소가 조건을 만족하지 않으면 false.
* 대량의 데이터를 처리할 때 특정 조건을 빠르게 확인.

const numbers = [1, 3, 5, 7];

const hasEven = numbers.some((num) => num % 2 === 0);

console.log(hasEven); // 출력: false (짝수가 없음)

**참고**

**'===' vs '=='의 차이**

* **===**: **엄격한 비교**로, 값과 데이터 타입이 모두 같아야 참(true).
* **==**: **느슨한 비교**로, 값이 같으면 참(true). (타입이 다르면 타입을 강제로 변환 후 비교)

**차이 예시:**

console.log(2 === '2'); // false (엄격 비교: 타입이 다름)

console.log(2 == '2'); // true (느슨한 비교: 문자열 '2'를 숫자 2로 변환)

**실무에서는 예상치 못한 오류를 방지하기 위해 ===를 사용하는 것이 더 안전.**  
==는 암묵적 타입 변환으로 의도치 않은 결과를 초래.

**1. Rest Parameter (나머지 매개변수)**

**Rest Parameter**는 함수에 전달되는 여러 개의 값을 **하나의 배열로 모아주는 기능**.  
쉽게 말해, 함수가 **얼마나 많은 값을 받을지 모를 때** 유용.

* 함수가 호출될 때 전달된 값(인자)을 배열로 처리 가능.
* ... (점 세 개)를 사용.

**예시 코드**

function sum**(...numbers)** {

**// numbers는 배열로 모든 인자를 받음**

return numbers.reduce((total, num) => total + num, 0);

}

console.log(sum(1, 2, 3)); // 6

console.log(sum(10, 20, 30, 40)); // 100

**한 줄 요약**

함수에 여러 값을 넘겨주면 Rest Parameter가 **배열로 묶어서 처리**!

**2. Arrow Function (화살표 함수)**

**Arrow Function**은 **간단하게 함수를 작성**할 수 있는 방법.  
일반적인 함수 선언보다 코드가 짧아지고 간결.

**설명**

* function 대신 **=> (화살표)**를 사용하여 함수 작성.
* 코드가 **짧아질** 뿐 아니라, **this** 키워드가 다른 함수와 다르게 작동해서 더 직관적.  
  즉, 일반 함수(function)과 달리 this를 고정.  
  this는 함수가 작성된 위치의 주인을 사용. (=헷갈리지 않음!)

**예시 코드**

// 일반 함수 선언

const add = function (a, b) {

return a + b;

};

// 화살표 함수로 동일한 기능 작성

const addArrow = (a, b) => a + b;

console.log(add(2, 3)); // 5

console.log(addArrow(2, 3)); // 5

**추가 예시 (1줄 함수)**

화살표 함수로 **한 줄**로 작성이 가능

const greet = (name) => `Hello, ${name}!`;

console.log(greet("Alice")); // Hello, Alice!

**요약**

* **Rest Parameter**: 여러 값을 함수로 전달할 때 **배열로 처리**.
* **Arrow Function**: 함수를 짧고 간단하게 **화살표(=>)로 작성**.

**참고**

**일반 함수에서의 this**

**예시 1**: 객체 안에서 함수 호출

const obj = {

name: "Alice",

sayName: function () {

console.log(this.name); // 'this'는 obj를 가리킴

}, };

obj.sayName(); // 출력: Alice

여기서 this는 obj를 가리켜. 왜냐하면 sayName()을 obj가 호출했기 때문!

**예시 2**: 그냥 호출하면?

function sayName() {

console.log(this.name); // 'this'는 전역 객체를 가리킴

}

sayName(); // 출력: undefined (브라우저에서는 window.name)

함수가 독립적으로 호출되면, this는 브라우저에서는 window, Node.js에서는 전역 객체.

[비교를 위한 추가자료]

**Arrow Function에서의 this**

Arrow Function은 this를 "고정"시킴  
일반 함수는 호출 방식에 따라 this가 바뀌지만, Arrow Function은 this가 작성된 위치의 주인을 사용.

**예시:** 일반 함수와 Arrow Function의 차이

const obj = {

name: "Alice",

sayName: function () {

const inner = () => {

console.log(this.name); // Arrow Function은 obj를 그대로 사용

};

inner();

},

};

obj.sayName(); // 출력: Alice

Arrow Function은 "이 함수가 어디에 작성되었는지"만 보고 this를 고정하기 때문에 **obj**를 가리킨다.

**1급 객체 (First-class Object)**

* JavaScript에서 **함수는 1급 객체**.
* 1급 객체란, **다른 값처럼 자유롭게 다룰 수 있는 객체**.

**이게 무슨 뜻일까?**

* 함수를 **변수에 저장**할 수 있다.
* 함수를 **다른 함수의 인자로 전달**할 수 있다.
* 함수를 **다른 함수의 반환값으로** 사용할 수 있다.

**예시 코드**

// 1. 함수는 변수에 저장 가능

**const sayHello = function () {**

console.log("Hello!");

};

sayHello(); // 출력: Hello!

// 2. 함수는 다른 함수의 인자로 전달 가능

function callFunction(**func**) {

func(); // 전달받은 함수 실행

}

callFunction(sayHello); // 출력: Hello!

// 3. 함수는 **다른 함수의 반환값**으로 사용 가능

function createGreeting() {

**return function ()** {

console.log("Hi there!");

};

}

const greeting = createGreeting();

greeting(); // 출력: Hi there!

**2. Object의 개념**

* **키-값 쌍으로 데이터를 저장**하는 구조.
* 사람의 정보를 저장하려고 할 때, 이름, 나이, 직업 등을 묶어서 하나로 표현하는 것이 객체.

**객체의 기본 구조**

* **키(key):** 데이터 이름.
* **값(value):** 데이터 내용.

**예시 코드**

// 객체 생성

const person = {

name: "Alice", // 키: name, 값: "Alice"

age: 25, // 키: age, 값: 25

job: "Developer", // 키: job, 값: "Developer"

};

// 객체 사용

console.log(person.name); // 출력: Alice

console.log(person.age); // 출력: 25

// 키-값 추가

person.country = "USA";

console.log(person.country); // 출력: USA

// 함수도 값이 될 수 있음

person.sayHello = function () {

console.log("Hello, I'm " + this.name);

};

person.sayHello(); // 출력: Hello, I'm Alice

**⭐요약**

1. **1급 객체:**
   * 함수도 숫자나 문자열처럼 **값처럼 사용 가능**.
   * 변수에 **저장, 전달, 반환**이 자유롭다.
2. **Object(객체):**
   * **키-값 쌍**으로 데이터를 저장하는 구조.
   * 다양한 데이터를 묶어서 **한 번에 관리**할 수 있다!!