DOM 조작하기 1

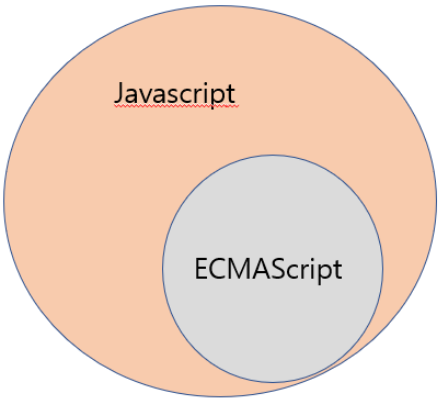
자료	Javascrtip
구분	Javascript
과목	

JavaScript 란?

JavaScript는 웹 브라우저에서 동작하는 유일한 프로그래밍 언어이다.
다른 언어들은 주로 애플리케이션 개발을 위한 도구로 설계되었지만,
JavaScript는 처음에 웹페이지의 보조 기능을 처리하기 위한 제한적인 용도로 만들어졌다.

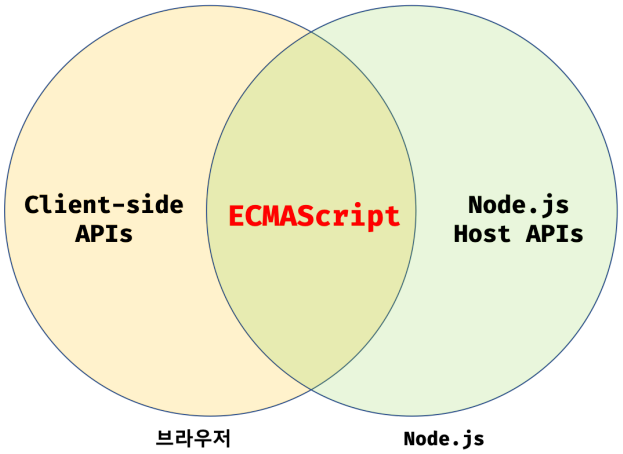
하지만 지금은 프론트엔드와 백엔드 영역을 모두 아우를 수 있는 강력한 개발 언어로 성장했다.
우리는 나중에 **Node.js**라는 JavaScript 실행 환경을 설치하게 될 것이다.
원래 JavaScript는 브라우저 엔진에서만 실행되었지만,
Node.js를 통해 브라우저 외의 환경, 즉 서버나 데스크탑 등에서도 JavaScript를 사용할 수 있게 된 것이다.

자, ECMAScript 가 무엇인지도 살펴보자.
JavaScript 표준 문법을 ECMAScript라고 할 수 있다.
ECMAScript는 JavaScript에서 값, 타입, 객체, 함수 등의 표준 문법 등을 규정한다.



JavaScript는 ECMAScript보다 상위 개념이다.
ECMAScript는 일반적인 프로그래밍 언어로서의 뼈대를 이루는 표준 문법이고,
JavaScript는 이 ECMAScript에 더해서 브라우저가 제공하는 Web API들(DOM, BOM 등)을 포함하는 개념이다.

즉, JavaScript는 **ECMAScript + Web API**를 아우르는 더 넓은 개념이라고 보면 된다.



JavaScript는 브라우저를 하나의 객체로 간주한다. 이 객체를 `window`라고 부른다.

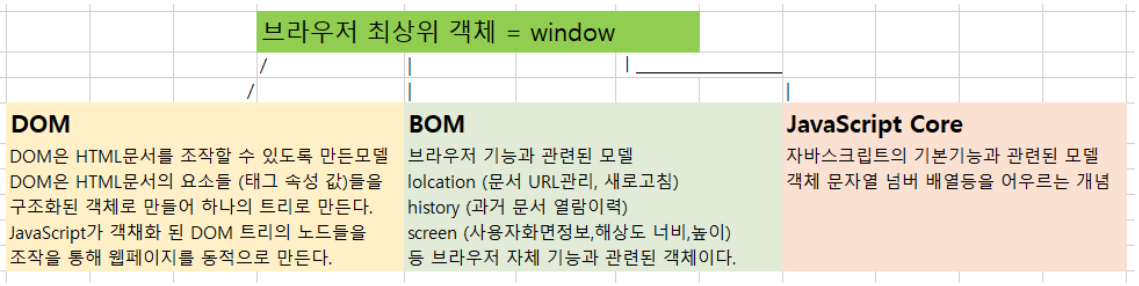
즉, 브라우저에서 JavaScript를 실행하면

모든 전역 변수와 함수들은 `window` 객체의 속성이나 메서드로 작동하게 된다.

이건 참고로 알아두면 좋다.

`window` 객체는 크게 세 가지로 나눌 수 있다.

바로 **DOM**, **BOM**, 그리고 **JavaScript Core** 이다.

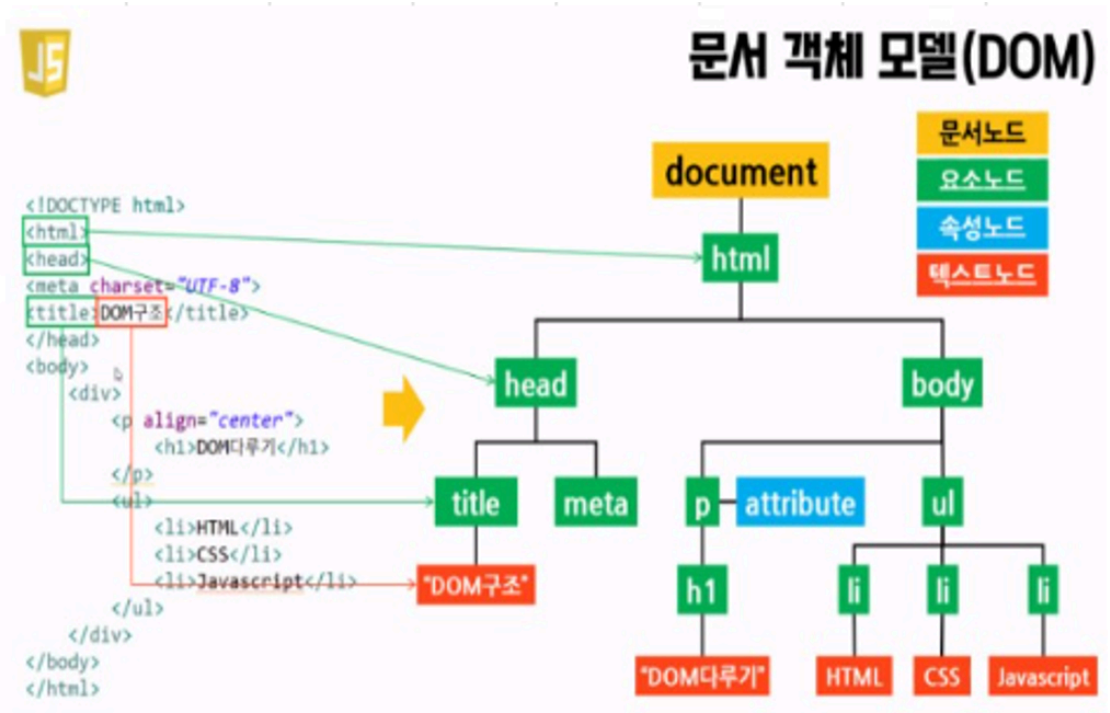


오늘 수업의 핵심은 **DOM을 조작하는 것**이다.

DOM은 JavaScript가 웹페이지에 접근해서 HTML 태그나 속성 같은 요소들을 조작할 수 있도록,

HTML 문서의 요소들을 **트리(tree)** 형태로 표현한 자료구조이다.

이걸 **DOM 트리**라고 부른다.



위 그림을 보면 DOM 트리는 부모-자식 관계로 이루어져 있다는 걸 확인할 수 있다.

DOM 트리의 최상단 루트 노드이자 객체를 `document`라고 부른다.

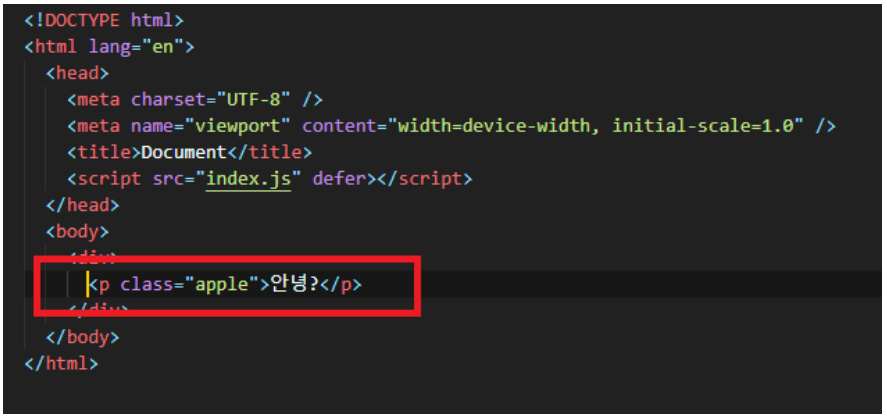
그 아래에 있는 상위 노드는 `<html>`이고, 그 밑에 `<head>`와 `<body>`가 자식 노드가 된다.

우리가 웹페이지를 방문하면, 웹페이지가 화면에 로딩되는 순서는 대략 이렇게 진행된다:

1. 웹 브라우저 실행
2. 브라우저가 웹 문서를 읽음 (`document`)
3. DOM 트리를 생성 → HTML 문서의 모든 요소를 객체 형태로 변환
4. 페이지 로딩 → CSS 스타일 적용 + 자바스크립트를 통한 동적 요소 반영
5. 최종적으로 화면에 표시

이런 식으로 웹페이지가 구성된다고 보면 된다.

DOM 트리를 구성하는 Node를 크게 3종류로 구분할 수 있다.



1. 요소노드 (Element Node)
    a. div 그리고 p와 같은 모든 HTML 태그를 의미하는 노드다
2. 속성노드 (Attribute Node)
    a. class id href 와 같이 HTML 태그의 속성(attribute)을 나타내는 노드다.
3. 텍스트노드 (Text Node)
    a. 안녕? 과 같은 텍스트 내용을 의미하는 노드다.

노드들을 트리의 구조로 나타내면 다음과 같다.

<p> (요소 노드)
├── class="apple" (속성 노드)
└── "안녕?" (텍스트 노드)

우리가 오늘 할 일은 JavaScript를 사용해서

DOM 객체들을 조작하는 것이라고 했다.

DOM 조작을 통해 웹페이지를 동적으로 바꿀 수 있기 때문이다.

조작하는 법은 간단하다.

1. 선택 후,
2. 조작하기

1. 선택 관련 메서드

`document.querySelector(selector)`

- element 한 개 선택
- 제공한 CSS selector를 만족하는 첫 번째 element 객체를 반환 (없다면 null반환)

`document.querySelectorAll(selector)`

- 일치하는 여러 element를 선택
- 제공한 CSSselector를 만족하는 NodeList를 반환

선택 관련 메서드 실습

```
<h1 id="title">DOM 조작</h1>
<p class="text">querySelector</p>
<p class="text">querySelectorAll</p>
<ul>
    <li>Javascript</li>
    <li>Python</li>
</ul>
```

```
console.log(document.querySelector('#title'))
// <h1 id="title">DOM 조작</h1>

console.log(document.querySelectorAll('.text'))
// NodeList(2) [p.text,    p.text]

console.log(document.querySelector('.text'))
// <p class="text">querySelector</p>

console.log(document.querySelectorAll('body > ul > li'))
// NodeList(2) [li,    li]

liTags = document.querySelectorAll('body > ul > li')
liTags.forEach(element => {
  console.log(element)
})
/*
<li>
    ::marker
    "Javascript"
</li>
<li>
    ::marker
    "Python"
</li>
*/
```

2. 조작 관련 메서드

생성 - `document.createElement('p')`

- `document.createElement('p')` 는 `<p>` **태그를 새로 만들고 반환하는 메서드**

- 예시:

```
const pTag = document.createElement('p');  // <p></p> 생성
```

입력 - `HTMLElement.innerText`

- `innerText`는 **DOM 요소 안에 있는 텍스트**를 가져오거나 변경
- 이때, **보이는 텍스트만** 취급하고, 스타일이나 숨겨진 내용은 무시함
- 예시:

```
const paragraph = document.querySelector('p');
console.log(paragraph.innerText);  // 화면에 보이는 텍스트만 출력
```

[참고] `node.textcontent` vs `HTMLElement.innerText`

innerText는 사용자에게 보이는 대로 출력, textcontent는 실제 데이터 출력

```
<p> node.textcontent vs HTMLElement.innerText </p>
```

```
const ptag = document.querySelector('p')
console.log(ptag.innerText)  // node.textcontent vs HTMLElement.innerText
console.log(ptag.textContent)  // node.textcontent vs HTMLElement.innerText

ptag.style.visibility = 'hidden'
console.log(ptag.innerText)  // 아무 것도 출력되지 않음
console.log(ptag.textContent)  // node.textcontent vs HTMLElement.innerText
```

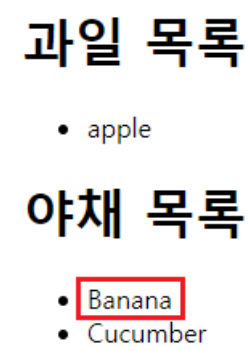
추가 - `Node.appendChild()`

- 한 Node를 특정 부모 Node의 자식 NodeList 중 마지막 자식으로 삽입
- 한번에 오직 하나의 Node만 추가할 수 있음
- 새롭게 생성한 Node가 아닌 이미 문서에 존재하는 Node를 다른 Node의 자식으로 삽입하는 경우, 위치를 이동

```
<h1>과일 목록</h1>
<ul id="fruits">
  <li id="apple">apple</li>
</ul>

<h1>야채 목록</h1>
<ul id="fruits">
  <li id="banana">Banana</li>
```

```
    <li id="cucumber">Cucumber</li>
</ul>
```

```
const fruitsList = document.querySelector('#fruits')

const banana = document.querySelector('#banana')

fruitsList.appendChild(banana)
```



삭제 - `Node.removeChild()`

- DOM에서 자식 Node를 제거
- 제거된 Node를 반환

조작 메서드 실습

```
// 태그 생성
const h1Tag = document.createElement('h1')

// 태그안에 컨텐츠를 작성하고
h1Tag.innerText = 'DOM 조작'

// 부모 div 태그를 가져와서
```

```
const div = document.querySelector('div')

// div 태그의 자식 요소로 추가
div.appendChild(h1Tag)

// div의 h1 요소 삭제
div.removeChild(h1Tag)
```

3. 속성 조회 및 설정

Element.getAttribute(속성이름)

- 해당 요소의 지정된 값(문자열)을 반환

Element.setAttribute(name, value)

- 지정된 요소의 값을 설정
- 속성이 이미 존재하면 값을 갱신, 존재하지 않으면 지정된 이름과 값으로 새 속성을 추가

속성 및 조회 실습

```
// a tag 생성 및 컨텐츠 추가
const aTag =  document.createElement('a')
aTag.innerText =   '구글'

// a 태그의 href 속성 추가
aTag.setAttribute('href', 'https://google.com')
console.log(aTag.getAttribute('href'))

// div 태그의 자식 태그로 a  태그 추가
const div = document.querySelector('div')
div.appendChild(aTag)

// h1 tag  선택 및 클래스 목록 조회
const h1 =  document.querySelector('h1')
console.log(h1.classList)  // DOMTokenList ['red', value: 'red']

// 클래스 추가
h1.classList.add('red')

// 클래스 제거
h1.classList.remove('red')

// 클래스가 존재한다면 제거하고 false를 반환,
// 존재하지 않으면 클래스를 추가하고 true를 반환
h1.classList.toggle('red')
h1.classList.toggle('blue')
console.log(h1.classList)  // DOMTokenList(2) ['red', 'blue', value: 'red blue']
```

[참고] 그 외 다양한 속성 조작 방법

- `Element.setAttribute(name, value)`
  - 해당 속성이 이미 존재하는 경우에는 갱신
  - 즉,새로운 값을 추가 또는 수정이 아닌,주어진 value로 새롭게 설정
  - 만약 기존 속성은 유지한 채로,새로운 값을 추가하고자 한다면
  - `Element.classList`, `Element.style` 등을 통해 직접적으로 해당 요소의 각 속성들을 제어 할 수 있음

실습 1

```
<ul id="list">
  <li class="red">apple</li>
  <li>orange</li>
  <li id="banana">banana</li>
  <li>grape</li>
  <li class="red">strawberry</li>
</ul>
```

- strawberry 아래에 새로운 과일 하나를 더 추가하기

```
// 추가할 과일 태그 만들기
const mango = document.createElement("li")
mango.innerText = '망고' // 새로 만든 li 태그에 망고 텍스트 넣어주기

const parent = document.querySelector("ul") // ul 태그 찾아오기(appendChild 쓰려면 부모 태그 찾아야 하니까..)
parent.appendChild(mango) // 찾은 부모 태그에 appendchild 이용해서 태그 추가하기
```

- apple 위에 새로운 과일을 하나 더 추가하기

```
const pineapple = document.createElement("li")
pineapple.innerText = '파인애플'

parent.prepend(pineapple)
```

- orange와 banana 사이에 새로운 과일을 하나 더 추가하기

```
const kiwi = document.createElement("li")
kiwi.innerText = '키위'

const banana = document.querySelector("#banana")

parent.insertBefore(kiwi, banana)
```

- class가 'red'인 노드 모두 삭제하기

```
const reds = document.querySelectorAll('.red')

reds.forEach(element => {
  element.remove()
});
```

- 모든 li 태그의 글자 크기를 키우기

```
const liTags = document.querySelectorAll('li')

liTags.forEach(element => {
    element.classList.add('big')
});
```

실습 2

```
// Apple, Banana, Orange 각각 li 요소 만들기
const fruits = ['Apple', 'Banana', 'Orange']

fruits.forEach(function () {
})

fruits.forEach((fruit) => {
  const li = document.createElement('li')
  li.innerText = fruit

  document.querySelector('#fruits-list').appendChild(li)
})
```

정리

```
선택 메서드
document.querySelector     요소 선택
document.querySelectorAll  해당요소 전부선택

요소 조작
document.createElement(태그이름)      태그 생성
부모.appendChild()                  자식요소 추가
부모.removeChile()                  자식요소 제거

속성 조작
.classList.add('test')          클래스속성 추가
.classList.remove('test')       클래스속성 값을 제거 (속성제거X)

(속성명 예> 'id' 'class' 등등)
```

```
.getAttribute('속성명')          속성값 가져오기
.setAttribute('속성명','값')     속성 그리고 값 설정/갱신
.removeAttribure('속성명')       속성 지우기

컨텐츠 조작
.textcontent ()                 컨텐츠 값 변경
```

[참고] NodeList는 배열이 아니다.

`document.querySelectorAll('p')`를 사용하면, **모든** `p` **태그를 선택**해서 `NodeList`라는 객체가 생성된다.

하지만 이 `NodeList`는 **배열**이 아니어서, 배열에서 제공하는 `map`, `filter`, `reduce` 같은 메서드는 사용할 수 없다는 점을 반드시 기억하자.

NodeList의 특징

1. **NodeList**는 **배열이 아닌 객체**로, 선택된 요소들을 나열한 유사 배열(like array)이다.
2. `forEach()` 메서드는 사용 가능:

   예시:

   ```
   const pTags = document.querySelectorAll('p');
   pTags.forEach(p => {
     console.log(p.textContent);  // 각 p 태그의 텍스트를 출력
   });
   ```

3. **배열 메서드 사용 불가**:

   `NodeList`에서는 `map`, `filter`, `reduce` 같은 메서드를 사용할 수 없다.

   ```
   const pTags = document.querySelectorAll('p');
   pTags.map(tag => tag.textContent);  // 에러 발생! map은 사용할 수 없음
   ```

따라서 `NodeList`는 배열은 다르지만, `NodeList`를 **배열로 변환**해서 사용할 수 있다.

- 배열 메서드를 사용하고 싶다면 `Array.from()` 을 이용해 `NodeList`를 배열로 변환할 수 있다.

  예시:

  ```
  const pTags = document.querySelectorAll('p');
  const pArray = Array.from(pTags);  // NodeList를 배열로 변환
  pArray.map(tag => tag.textContent);  // 이제 배열 메서드 사용 가능하지만 우리 아직 map학습 안함
  ```

끝