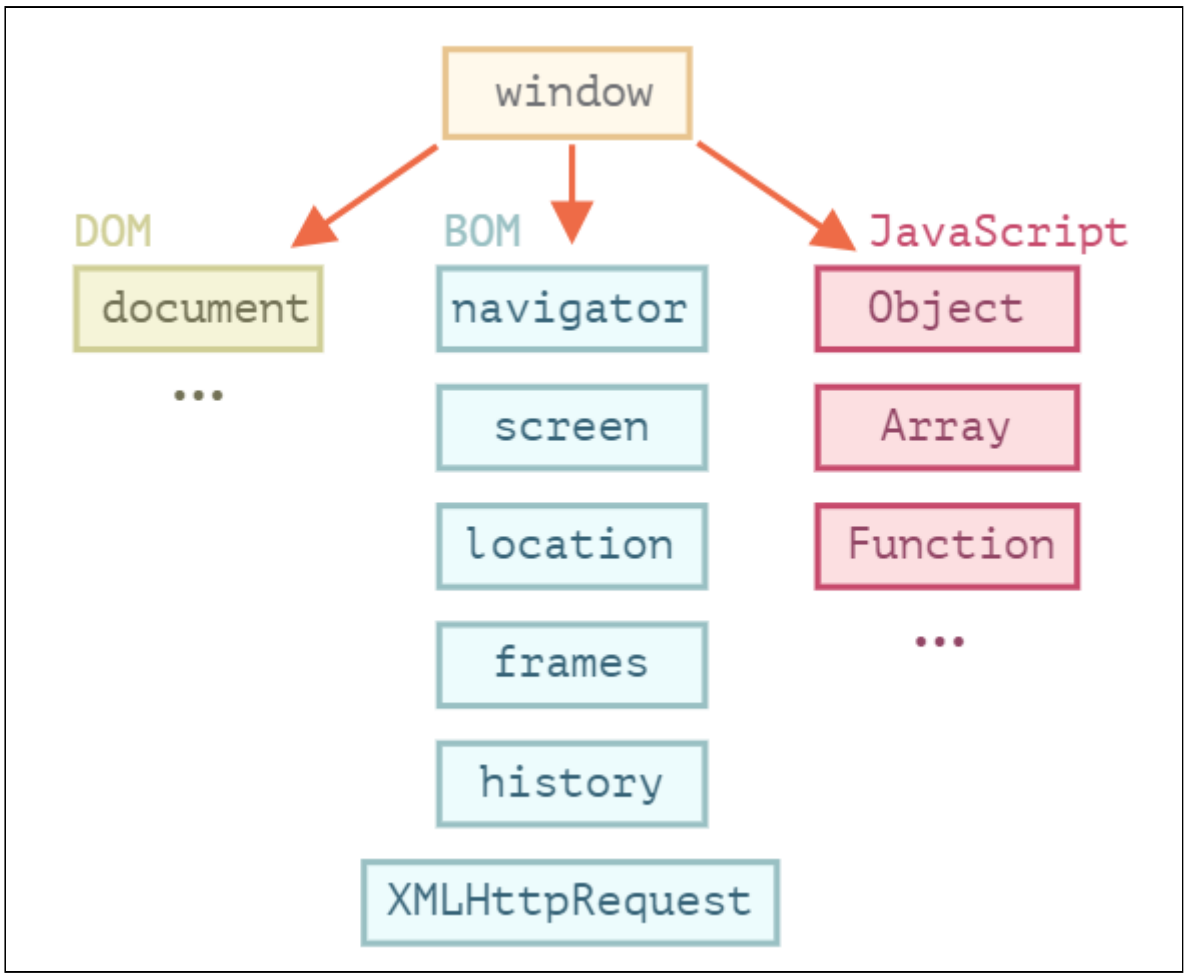
문서 1

브라우저 환경과 다양한 명세서

- 자바스크립트는 본래 **웹 브라우저**에서 사용하려고 만든 언어
  - https://www.bloger.kr/58
- 이후 진화를 거쳐 **다양한 사용처와 플랫폼을 지원하는 언어**로 변모함
- 자바스크립트가 돌아가는 플랫폼을 **호스트(host)**라고 부름
  - 각 플랫폼은 **해당 플랫폼에 특정되는 기능을 제공**하는데, 자바스크립트 명세서에선 이를 **호스트 환경(host environment)**이라고 칭함
  - 즉, 랭귀지 코어(ECMAScript)에 더하여 **플랫폼에 특정되는 객체와 함수**를 제공함
    - 웹브라우저는 **웹페이지를 제어하기 위한 수단(DOM, BOM)을 제공**하고, Node.js는 서버 사이드와 관련된 기능(ex: 파일 처리, 네트워크 요청 등)을 제공



호스트 환경이 웹 브라우저일 때 사용할 수 있는 기능

- 브라우저 환경인 경우 최상단엔 **window라고 불리우는 루트 객체**가 있으며 window 객체는 두 가지 역할을 수행함
  - 자바스크립트 코드의 전역 객체 역할
  - 브라우저 창(browser window)을 대변하고, 이를 제어할 수 있는 **객체와 메서드를 제공**하는 역할 (DOM, BOM)

- 가령, alert 메서드는 "브라우저 환경"에서 경고창을 띄우는 용도로 설계된 메서드로 Node.js에서는 사용이 불가함

문서 객체 모델(DOM)

- **문서 객체 모델(Document Object Model, DOM)**은 웹 페이지 내의 모든 콘텐츠를 **객체로 나타내어** 읽기 작업과 내용 수정이 가능하도록 함
- DOM을 조작할 수 있도록 **document 객체**가 주어지며 이 객체는 페이지의 기본 진입점 역할을 수행함
  - document 객체를 이용해 문서 내부의 어떤 요소든 변경할 수 있고, 필요한 요소를 생성하여 동적으로 변경도 가능함

브라우저 객체 모델(BOM)

- 브라우저 객체 모델(Browser Object Model, BOM)은 **문서 이외의 모든 것을 제어**하기 위해 브라우저(호스트 환경)가 제공하는 추가 객체
  - navigator, location, screen, storage와 같은 객체가 제공되며 window 객체 또한 넓게 보면 BOM에 포함된다고 볼 수 있음
- alert, confirm, prompt 메소드 역시 BOM의 일부

BOM 객체 중 하나인 location 객체 활용 사례

```
alert(location.href); // 현재 URL을 보여줌
if(confirm("위키피디아 페이지로 가시겠습니까?")) {
    location.href = "https://wikipedia.org"; // 새로운 페이지로 넘어감
}
```

DOM 트리

- 문서 객체 모델(DOM)에 따르면 **문서 내의 모든 HTML 태그(=요소)는 객체로 표현**됨
- 태그 내의 문자(text) 역시 모두 객체로 표현됨
- **모든 객체는 자바스크립트를 통해 접근**할 수 있고, 페이지의 내용을 조작할 때 이 객체를 사용함

문서의 배경 색을 바꾸는 코드

```
document.body.style.background = 'red';
```

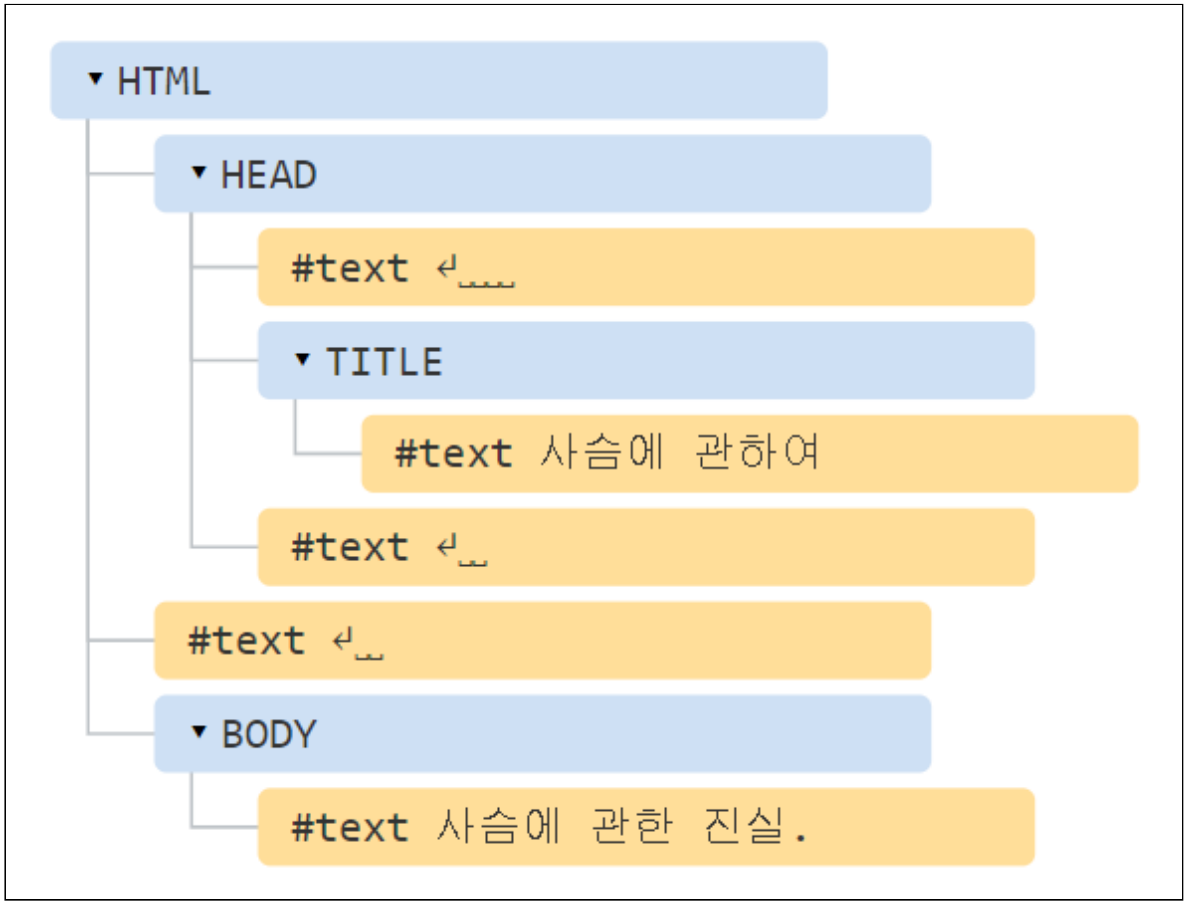
dom_structure.html

```
<!DOCTYPE HTML>
<html>
<head>
    <title>사슴에 관하여</title>
</head>
<body>
    사슴에 관한 진실.
</body>
</html>
```

개발자 도구 콘솔 창 입력

```
function blankTextTo(str, enter="(엔터)", space="(스페이스)", tab="탭") {
    return str.replaceAll('\n', enter).replaceAll(' ', space).replaceAll('\t',
tab);
}
console.log(document.head);
console.log(document.head.childNodes);
console.log(blankTextTo(document.head.childNodes[0].nodeValue));
console.log(document.head.childNodes[1]);
console.log(document.head.childNodes[1].childNodes[0].nodeValue); // 사슴에 관하여
console.log(blankTextTo(document.head.childNodes[2].nodeValue));
console.log(blankTextTo(document.body.childNodes[0].nodeValue)); // 사슴에 관한 진
실.
```

- 위의 HTML의 구조 나타내기



- 문서 트리 내의 모든 내용들은 모두 **노드(node)**로 표현됨
  - 노드는 모두 객체를 통해서 접근할 수 있음
- 태그는 **요소 노드(element node)**로 표현됨
- 문자는 모두 **텍스트 노드(text node)**로 표현됨
  - 텍스트 노드는 문자열 내용만 담을 수 있고 자식 노드를 가질 수 없으며 잎 노드(leaf node)가 됨
- 공백이 없는 텍스트 노드만으로 HTML 문서를 구성하려면 HTML을 아래와 같이 만들어야 함

```
<!DOCTYPE HTML>
<html><head><title>사슴에 관하여</title></head><body>사슴에 관한 진실.</body></html>
```

- 기형적인 HTML을 만나면 브라우저는 **DOM 생성 과정에서 HTML을 자동으로 교정**함

    - 즉, DOM 생성 과정에서 브라우저는 문서에 있는 에러(ex: html, head, body 태그가 없음, 닫는 태그가 없음 등)를 자동으로 처리함

기타 노드 타입

- 요소와 텍스트 노드 외에도 다양한 노드 타입이 존재함 (ex: 주석 노드)
    - 주석 노드는 **HTML에 포함된다면 반드시 DOM 트리에도 추가되어야 한다는 규칙** 때문에 존재함
    - HTML 안의 모든 내용들은 (비록 그것이 주석이더라도) DOM을 구성함
        - HTML 문서 최상단에 위치하는 <!DOCTYPE...> 지시자 또한 DOM 노드가 됨

실무에선 주로 다루는 네 가지 노드 유형 (그렇지만 보통은 document와 요소 노드만 다룸)

```
1. DOM의 "진입점"이 되는 문서(document) 노드
2. HTML 태그에서 만들어지며, DOM 트리를 구성하는 블록인 요소 노드(element node)
3. 텍스트를 포함하는 텍스트 노드(text node)
4. 화면에 보이지는 않지만, 정보를 기록하고 자바스크립트를 사용해 이 정보를 DOM으로부터 읽을 수 있는 주석(comment) 노드
```

- 실제로 존재하는 노드는 12가지

DOM 탐색하기 (***)

- 문서 내용에 변경을 가하기 전에 **조작하고자 하는 DOM 객체에 접근하는 것이 선행**되어야 함
- DOM에 수행하는 모든 연산은 **document 객체에서 시작**하게 되며 해당 객체를 통해 어떠한 요소에도 접근 가능함
- DOM 트리 상단의 노드들은 document가 제공하는 프로퍼티를 사용해 접근

```
<html> => document.documentElement
<body> => document.body
<head> => document.head
```

- DOM에서 **null 값은 "해당하는 노드가 없음"을 의미**

childNodes, firstChild, lastChild 속성으로 자식 노드 탐색하기

- **자식 노드(child node, children)**는 바로 아래의 자식 요소를 의미
- **후손 노드(descendants)**는 중첩 관계에 있는 모든 요소를 의미 (자식 노드, 자식 노드의 모든 자식 노드 등이 후손 노드)
- **childNodes 컬렉션**은 텍스트 노드를 포함한 모든 자식 노드를 담고 있는 콜렉션임
- **firstChild와 lastChild 프로퍼티**를 이용하면 첫 번째, 마지막 자식 노드에 빠르게 접근 가능함

dom_traverse.html

```
<body>
    <div>
        <h1>제목</h1>
        <ul>
            <li>항목 1</li>
            <li>항목 2</li>
            <li>항목 3</li>
        </ul>
        <p>단락</p>
    </div>
</body>
```

개발자 도구 콘솔 창 입력

```
// 텍스트 노드에 주의하며 속성에 의해서 반환될 값 유추해보기
let div = document.body.childNodes[1];
console.log(div.childNodes);
console.log(div.childNodes[0]); // h1이 아님을 유의!
console.log(div.firstChild === div.childNodes[0]);
console.log(div.childNodes[1]); // h1
console.log(div.childNodes[3]); // ul
console.log(div.childNodes[5]); // p
console.log(div.lastChild === div.childNodes[6]);
```

DOM 컬렉션

- childNodes는 배열이 아닌 **반복 가능한(iterable) 유사 배열 객체**인 컬렉션(collection)
- 배열은 아니어도 반복 가능한 객체이므로 for .. of 반복문이나 forEach 메서드를 통해서 순회 가능
    - 단, 배열 관련 메서드(push, map, filter 등)는 사용 불가

형제와 부모 노드

- 같은 부모를 가진 노드를 **형제 노드(sibling node)**라고 부름
- 다음 형제 노드는 **nextSibling 속성**을 통해서, 이전 형제 노드에는 **previousSibling 속성**을 통해 접근
- 부모 노드에 대한 정보는 **parentNode 속성**을 통해 접근 가능

개발자 도구 콘솔 창 입력

```
let h1 = document.body.childNodes[1].childNodes[1];
console.log(h1.parentNode); // div
console.log(h1.previousSibling); // null이 아니고 텍스트 노드임을 유의
console.log(h1.previousSibling.previousSibling); // null
console.log(h1.nextSibling); // ul이 아니고 텍스트 노드임을 유의
console.log(h1.nextSibling.nextSibling); // ul
let p = h1.nextSibling.nextSibling.nextSibling.nextSibling;
console.log(p.nextSibling); // null이 아니고 텍스트 노드임을 유의
console.log(p.nextSibling.nextSibling); // null
```

요소 노드에만 적용 가능한 속성 살펴보기

- **parentElement, children, firstElementChild, lastElementChild, previousElementSibling, nextElementSibling 속성**은 노드와 관련된 속성과 비슷하게 작동하지만 **오직 요소만을 대상으로 한다는 점**에서 차이가 있음
- 실질적으로는 **요소와 관련된 속성이 더 많이 사용**됨

개발자 도구 콘솔 창 입력

```
let div = document.body.firstElementChild;
console.log(div.children);
console.log(div.children[0]); // h1
console.log(div.firstElementChild === div.children[0]); // true
console.log(div.children[1]); // ul
console.log(div.children[2]); // p
console.log(div.lastElementChild === div.children[2]); // true
let h1 = div.children[0];
console.log(h1.previousElementSibling); // null
console.log(h1.nextElementSibling); // ul
console.log(h1.parentElement); // div
```

- 거의 모든 **DOM 컬렉션은 DOM의 현재 상태를 반영**함
    - 즉 특정 컬렉션을 참조하고 있는 도중에 DOM에 새로운 노드가 추가되거나 삭제될 경우에도 변경사항이 컬렉션에도 자동으로 반영

dom_livedata.html

```
<body>
    <ul>
        <li>항목 1</li>
        <li>항목 2</li>
        <li>항목 3</li>
    </ul>
</body>
<script>
let ul = document.body.firstElementChild;

// 3초마다 ul에 새로운 li 요소를 추가
setInterval(function() {
    let newListItem = document.createElement('li');
    newListItem.appendChild(document.createTextNode('항목 ' + (ul.children.length
+ 1)));
    ul.appendChild(newListItem);
}, 3000);

setInterval(function() {
    // ul의 자식들 개수가 계속해서 변경됨 (데이터가 살아있음)
    console.log('count: ' + ul.children.length);
}, 1000);
</script>
```

테이블 탐색하기 [TODO]

- 일부 DOM 요소 노드는 편의를 위해 기본 프로퍼티 외에 추가적인 프로퍼티를 지원함

요약

- 탐색 속성은 크게 두 개의 집합으로 나뉨 (실질적으로는 요소와 관련된 속성이 더 많이 사용됨)

```
모든 노드에 적용 가능한 집합
parentNode, childNodes, firstChild, lastChild, previousSibling, nextSibling

(*) 요소 노드에만 적용 가능한 집합
parentElement, children, firstElementChild, lastElementChild,
previousElementSibling, nextElementSibling
```

```
문제 풀기
exercises/lec01
- exercise01.html
- exercise02.html
```

getElement*, querySelector*로 요소 검색하기 (****)

- 상대 위치를 이용하지 않으면서 웹 페이지 내에서 원하는 요소 노드에 바로 접근하는 방법 알아보기

document.getElementById 혹은 id를 사용해 요소 검색하기

- 요소에 **id 속성**이 있으면 위치에 상관없이 document 객체의 **getElementById 메서드**를 이용해 요소에 접근 가능

```
// 요소 얻기
let elem = document.getElementById('elem');

// 배경색 변경하기
elem.style.background = 'red';
```

- 요소에 id 속성값을 부여시, **id 속성값을 그대로 딴 전역 변수가 생성**되므로 이를 이용해 요소에 접근 가능 (권장 X)
    - 요소 id를 따서 자동으로 선언된 전역변수는 동일한 이름을 가진 변수가 선언되면 이후에는 접근이 불가능함

```
// 변수 elem은 id가 'elem'인 요소를 참조합니다.
elem.style.background = 'red';

// 전역 변수에 접근하여 다른 값을 덮어씌우기
elem = 100;
// 이후 접근 불가
// elem.style.background = 'yellow';

// id가 elem-content인 요소는 중간에 하이픈(-)이 있기 때문에 변수 이름으로 쓸 수 없으므로, 대괄호(`[...]`)를 사용해서 window['elem-content']로 접근
```

- id는 유일해야 하며, **문서 내 요소의 id 속성값은 중복되어선 안 됨**을 유의
    - 만약 똑같은 id를 가진 요소가 여러 개 있으면 getElementById와 같이 id를 이용해 요소를 검색하는 메서드의 동작 예측이 불가능해짐
        - 보통은 가장 첫 번째로 만나게 되는 id 요소를 반환하게 되지만 이는 어디까지나 브라우저의 구현 방식에 따라 달라짐

querySelectorAll, querySelector, matches, closest

- **선택자를 이용하여 요소를 선택**하는 메소드들도 존재하며 **메서드의 인자로 문자열 형식의 선택자를 전달**
- **querySelectorAll 메서드**는 자식 요소 중 주어진 CSS 선택자에 대응하는 **모든 요소를 반환**
- **querySelector 메서드**는 주어진 CSS 선택자에 대응하는 요소 중 **첫 번째 요소를 반환**
    - 즉, `querySelectorAll(css)[0]`과 동일하게 작동
- matches 메서드는 **주어진 CSS 선택자를 이용해서 해당 요소를 선택할 수 있는지 여부를 판단**하며 일치하면 true, 아니라면 false를 반환함
    - 요소가 담겨있는 배열 등을 순회해 원하는 요소만 걸러내고자 할 때 유용하게 사용 가능
- closest 메서드는 **요소 자기 자신을 포함**하여 CSS 선택자와 일치하는 **가장 가까운 조상 요소**를 찾아서 반환함
    - closest메서드는 해당 요소부터 시작해 DOM 트리를 한 단계씩 거슬러 올라가면서 원하는 요소를 찾고 CSS 선택자와 일치하는 요소를 찾으면 검색을 중단하고 해당 요소를 반환함 (즉, 가장 먼저 발견한 요소를 반환함)

query_using_selector_1.html

```
<body>
    <h1 id="title">title</h1>
    <ul class="my-list">
        <li>first</li>
        <li>second</li>
        <li>third</li>
    </ul>
    <div>
        <h2>h2 #1</h2>
        <div>
            <h2>h2 #2</h2>
            <div>
                <h2>h2 #3</h2>
            </div>
        </div>
    </div>
```

```
</body>
<script>
// 문자열로 된 선택자 정보를 메서드 인자값으로 전달
let title = document.querySelector('#title');
let ul = document.querySelector('ul.my-list');
// ul "요소 내부"에서 해당 선택자를 이용해 검색
let lastListItem = ul.querySelector('li:last-child');
// 선택자를 더 길게 써서 document를 통해서도 접근 가능
lastListItem = document.querySelector('ul.my-list li:last-child');
let nonExist = document.querySelector('#non-exist'); // 존재하지 않으면 null 반환
console.log(title, ul, lastListItem, nonExist);

// querySelectorAll은 콜렉션을 반환함
let allListItem = ul.querySelectorAll('li');
let allH2 = document.querySelectorAll('h2');
console.log(allListItem, allH2);
</script>
```

query_using_selector_2.html

```
<body>
    <div class="my-div">
        <h2>h2 #1</h2>
        <div>
            <h2>h2 #2</h2>
            <div>
                <h2>h2 #3</h2>
            </div>
        </div>
    </div>
</body>
<script>
let myDiv = document.querySelector('div.my-div');
let allDescendents = myDiv.querySelectorAll("*");
for(let d of allDescendents) {
    // matches로 전달한 선택자로 해당 요소를 선택할 수 있는지 여부를 확인 가능
    console.log(d, d.matches("div.my-div h2:first-child"));
}
/*
for(let d of allDescendents) {
    console.log(d, d.matches("div.my-div h2:last-child"));
}
*/

let lastH2 = myDiv.querySelector("div.my-div h2:last-child");
let div1 = lastH2.closest("div");
let div2 = lastH2.closest("div.my-div");
console.log(div1, div2);
// h2 요소에서 가장 먼저 만나는 h2 요소는 자기 자신 (검색이 자기 자신부터 수행됨을 유의)
console.log(lastH2.closest("h2") == lastH2); // true
</script>
```

getElementsBy* (***)

- 태그 종류나 클래스 이름을 이용해 원하는 노드를 찾아주는 메서드도 존재함
    - 단, CSS 선택자를 전달하는 메서드(ex: querySelector)를 이용하는게 더 편리하고 문법도 짧기 때문에 요즘에는 잘 사용되지는 않음 (레가시 코드에서 발견 가능)
- elem.getElementsByTagName(tag)
    - 요소 내부에서 주어진 태그에 해당하는 요소를 찾고, 대응하는 요소를 담은 컬렉션을 반환
- 매개변수로 "*"이 전달되면 모든 태그가 반환됨
- elem.getElementsByClassName(className)
    - 요소 내부에서 주어진 class 이름을 기준으로 요소를 찾고, 대응하는 요소를 담은 컬렉션을 반환
- document.getElementsByName(name)
    - 아주 드물게 쓰이는 메서드로, 문서 전체를 대상으로 검색을 수행하며 검색 기준은 name 속성값이고, 이 메서드 역시 검색 결과를 담은 컬렉션을 반환
- 위 메서드들은 모두 **요소 하나가 아닌, 컬렉션을 반환함**을 유의
- "getElementsBy"로 시작하는 모든 메서드는 살아있는 컬렉션을 반환
- 즉, 문서에 변경이 있을 때마다 **컬렉션이 자동으로 갱신되어 최신 상태를 유지**함
- querySelectorAll 메서드는 **정적인 컬렉션을 반환**하므로 호출 시점의 컬렉션 내용을 그대로 유지함

get_elements_by.html

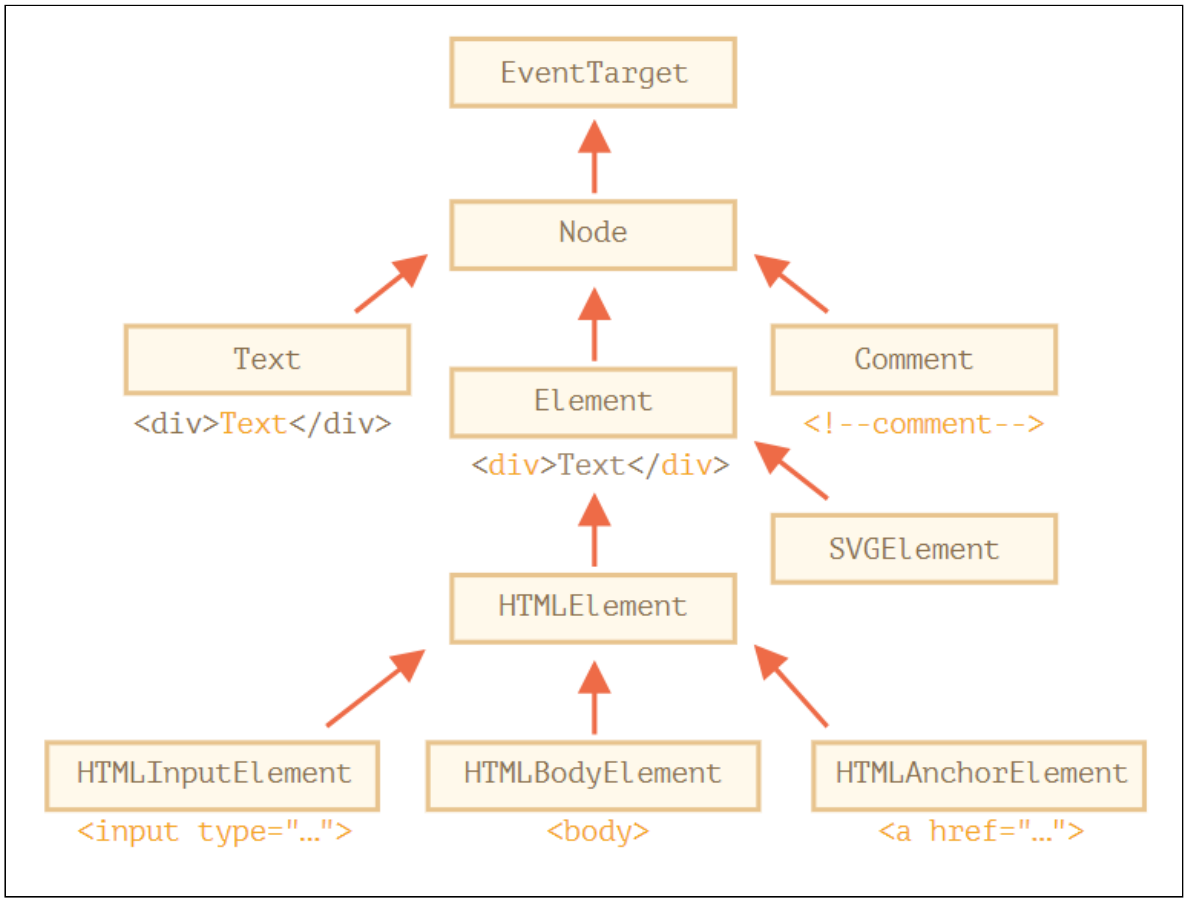
```
<body>
    <form>
        <label>이름</label><input type="text" name="name" /><br />
        <label>패스워드</label><input type="password" name="password" />
    </form>
    <div class="my-div">my div</div>
</body>
<script>
// 태그 이름을 통해서 요소 검색
let form = document.getElementsByTagName("form")[0];
let inputs = form.getElementsByTagName("input");
let body = document.getElementsByTagName("body")[0];

// 클래스 이름을 통해서 요소 검색
let myDiv = body.getElementsByClassName("my-div")[0];
console.log(form, inputs, body, myDiv);

// name 속성을 대상으로 요소 검색 (document 객체를 통해서만 메서드 호출 가능)
let nameInput = document.getElementsByName("name")[0];
let passwordInput = document.getElementsByName("password")[0];

console.log(nameInput, passwordInput);
</script>
```

```
문제 풀기
exercises/lec01
- exercise03.html
```

주요 노드 프로퍼티

DOM 노드 클래스

- DOM 노드는 종류에 따라 각각 다른 속성을 가지고 있음
  - 가령, a 태그에 대응하는 요소 노드엔 링크 관련 속성을, input 태그에 대응하는 요소 노드엔 입력 관련 속성을 제공함
- DOM 노드는 **공통 조상**으로부터 만들어지기 때문에 노드 종류는 다르지만, 모든 DOM 노드는 공통된 속성과 메서드를 지원



- nodeType 프로퍼티는 **DOM 노드의 타입**을 알아내고자 할 때 쓰이는 구식 속성
  - https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType

```
elem.nodeType == 1 - 요소 노드
elem.nodeType == 3 - 텍스트 노드
elem.nodeType == 9 - 문서 객체
```

- **nodeName이나 tagName 속성**을 통해서 DOM 노드의 "태그 이름"을 알아낼 수 있음
  - nodeName은 모든 노드에서 지원되지만, tagName은 Element 클래스로부터 유래되었기 때문에 요소 노드에서만 지원함
  - 만약, 요소 노드만 다루고 있다면 tagName과 nodeName에는 차이가 없으므로 둘 다 사용 가능함

dom_node_class_1.html

```
<body>
    <a href="http://www.naver.com">naver</a>
    <input type="text" value="Hello" />
    <p>paragraph</p>
    <!-- 주석 -->
</body>
<script>
let a = document.body.children[0];
let input = document.body.children[1];
let p = document.body.children[2];
let pText = p.childNodes[0];
let comment = document.body.childNodes[7];

console.log(a.nodeType, input.nodeType, p.nodeType); // 요소이므로 전부 1
console.log(pText.nodeType); // 텍스트 노드이므로 3
console.log(comment.nodeType); // 주석 노드이므로 8
console.log(document.nodeType); // 진입점 역할을 하는 문서 노드이므로 9

// 요소 노드의 경우 nodeName, tagName 값의 차이가 없으므로 모두 "A"
console.log(a.nodeName, a.tagName);
// 요소 노드가 아닌 경우 nodeName만 값이 존재하고 tagName은 undefined
console.log(pText.nodeName, pText.tagName); // nodeName => #text
console.log(comment.nodeName, comment.tagName); // nodeName => #comment
</script>
```

- **innerHTML 속성**을 사용하면 요소 안의 HTML을 **문자열 형태**로 반환받을 수 있음
  - 해당 속성에 += 연산자를 쓸 경우 추가가 아니라 내용을 덮어쓰기 때문에 주의해서 사용해야 함
  - += 연산자를 사용할 경우, **기존 내용이 완전히 삭제**된 후 밑바닥부터 다시 내용이 쓰여지기 때문에 이미지 등의 리소스 전부가 다시 로딩됨을 유의
- innerHTML 속성은 **요소 노드에만 사용**할 수 있음
  - 텍스트 노드 같은 다른 타입의 노드에는 innerHTML과 유사한 역할을 해주는 속성인 **nodeValue와 data를 사용**함
- **textContent 속성**을 사용하면 요소 내의 텍스트를 추출할 수 있으며 이 과정에서 **태그와 관련된 내용은 제외**됨
  - textContent를 사용하면 텍스트를 "안전한 방법"으로 쓸 수 있기 때문에 실무에선 textContent를 쓰기 용으로 유용하게 사용
    - 가령, XSS 공격을 예방하기 위해서 사용 가능

```
- innerHTML을 사용하면 사용자가 입력한 문자열이 "HTML 형태"로 태그와 함께 저장됨
- textContent를 사용하면 사용자가 입력한 문자열이 "순수 텍스트 형태로" 저장되기 때문에 태그를 구성하는 특수문자들이 문자열로 처리됨
```

XSS 공격 예시 코드

```
let inner = prompt("문서 내용 입력", `<img src='x' onerror='for(let i=0;i<10;i++) { alert("Hello"); }'>`)
document.body.innerHTML = inner;
```

- outerHTML 프로퍼티엔 요소 전체 HTML 내용이 담겨 있음
- hidden 속성은 **요소를 보여줄지 여부를 지정**할 때 사용함

- hidden 속성값을 true로 설정하는 것은 `display:none` 스타일을 적용하는 것과 동일한 역할을 함

dom_node_class_2.html

```
<body>
    <p>안녕하세요. <b>반갑습니다.</b> <i>이탤릭체</i>로 표시한 글자</p>
    <div>
        <ul>
            <li>항목1</li>
            <li>항목2</li>
        </ul>
    </div>
    <span>텍스트만 있는 스팬 요소</span>
    <b>b</b>
    <button>button</button>
</body>
<script>
let p = document.body.children[0];
let div = document.body.children[1];
let span = document.body.children[2];

// 태그 요소가 내용에 포함된 텍스트 추출
console.log(p.innerHTML);
console.log(div.innerHTML);
console.log(span.innerHTML);

// 태그 요소가 제거된 텍스트 추출
console.log(p.textContent);
console.log(div.textContent);
// 애초에 내부에 태그 요소가 하나도 없었으므로 innerHTML과 결과가 동일함
console.log(span.textContent);

let spanText = span.childNodes[0];
// 텍스트 노드에서 내용(텍스트)에 접근하려면 nodeValue나 data 속성을 사용
console.log(spanText.nodeValue);
console.log(spanText.data);

// innerHTML 값을 조작하여 새로운 내용으로 변경 가능 (태그 내용도 포함 가능)
p.innerHTML = "<b>볼드체</b>로 쓰여진 <strike>새로운</strike> 내용";
// innerHTML과는 다르게 태그와 관련된 내용이 escape 처리되어 나타남
// p.textContent = "<b>볼드체</b>로 쓰여진 <strike>새로운</strike> 내용";
// += 연산자를 사용할 경우 기존 내용에 덧붙여서 내용이 추가되긴 하지만, 내부적으로는 아예
기존의 내용을 모두 삭제하고 새로 구성하게 된다는 점을 유의
// p.innerHTML += "<b>볼드체</b>로 쓰여진 <strike>새로운</strike> 내용";

let b = document.body.children[3];
// outerHTML을 "해당 요소(b)를 포함"한 요소 내용을 반환함 (해당 요소 + 자식 요소)
console.log(b.outerHTML);
// innerHTML과 마찬가지로 내용 수정 가능
b.outerHTML = "<strike>strike</strike>";
// 단, 이후 b를 통해서는 수정된 내용을 읽어올 수 없음
console.log(b.outerHTML);
// 이후 내용을 수정하는 것도 불가능함을 유의
// b.outerHTML = "<strike>strike again</strike>";
// 수정된 내용에 접근하기 위해서는 새로 노드 객체를 반환받아야 함
let strike = document.body.children[3];
```

```
console.log(strike.outerHTML);

let btn = document.body.children[4];
setInterval(function() {
    // hidden 속성값을 true로 주면 화면에서 보이지 않게 됨 (display: none)
    btn.hidden = !btn.hidden;
}, 1000);
</script>
```

```
문제 풀기
exercises/lec01
- exercise04.html
```