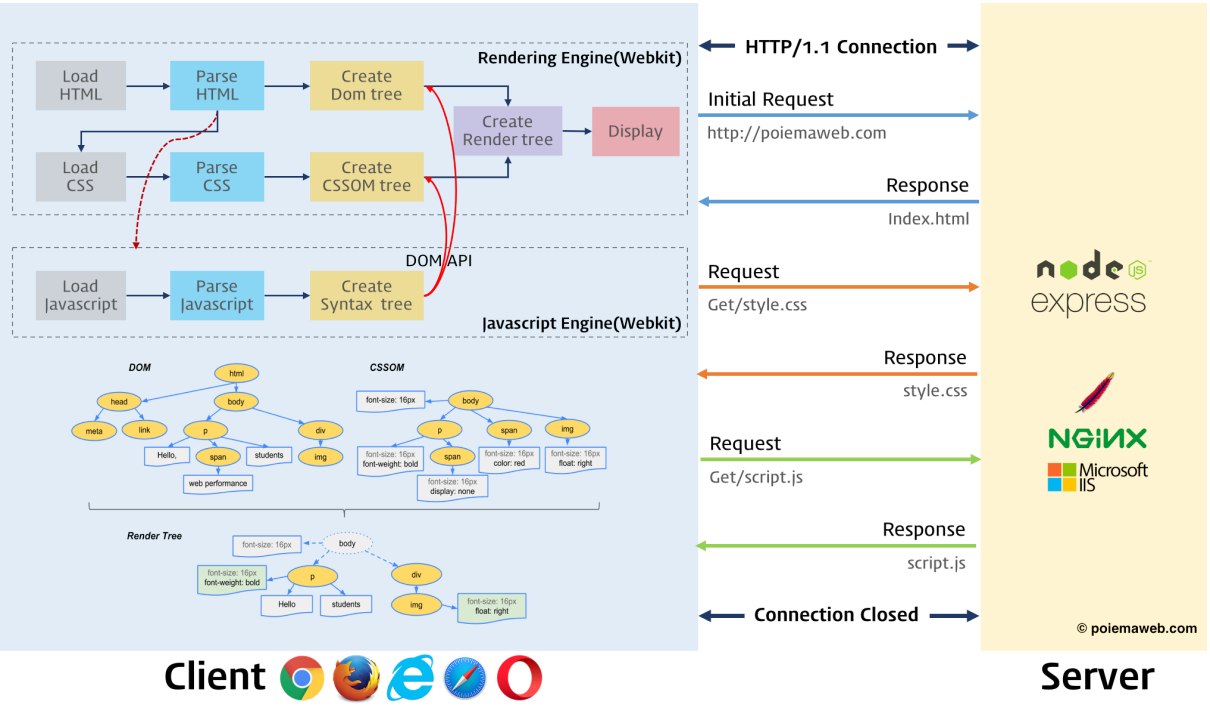
문서 객체 모델(Document Object Model)

DOM (Document Object Model)

텍스트 파일로 만들어져 있는 웹 문서를 브라우저에 렌더링하려면 웹 문서를 브라우저가 이해할 수 있는 구조로 메모리에 올려야 한다. 브라우저의 렌더링 엔진은 웹 문서를 로드한 후, 파싱하여 웹 문서를 브라우저가 이해할 수 있는 구조로 구성하여 메모리에 적재하는데 이를 DOM이라 한다. 즉 모든 요소와 요소의 어트리뷰트, 텍스트를 각각의 객체로 만들고 이들 객체를 부자 관계를 표현할 수 있는 트리 구조로 구성한 것이 DOM이다. 이 DOM은 자바스크립트를 통해 동적으로 변경할 수 있으며 변경된 DOM은 렌더링에 반영된다.



브라우저는 웹 문서(HTML, XML, SVG)를 로드한 후, 파싱하여 DOM(문서 객체 모델: Document Object Model)을 생성한다.

이러한 웹 문서의 동적 변경을 위해 DOM은 프로그래밍 언어가 자신에 접근하고 수정할 수 있는 방법을 제공하는데 일반적으로 프로퍼티와 메소드를 갖는 자바스크립트 객체로 제공된다. 이를 DOM API(Application Programming Interface)라고 부른다. 달리 말하면 정적인 웹페이지에 접근하여 동적으로 웹페이지를 변경하기 위한 유일한 방법은 메모리 상에 존재하는 DOM을 변경하는 것이고, 이때 필요한 것이 DOM에 접근하고 변경하는 프로퍼티와 메소드의 집합인 DOM API이다.

DOM은 HTML, ECMAScript에서 정의한 표준이 아닌 별개의 W3C의 공식 표준이며 플랫폼/프로그래밍 언어 중립적이다. DOM은 다음 두 가지 기능을 담당한다.

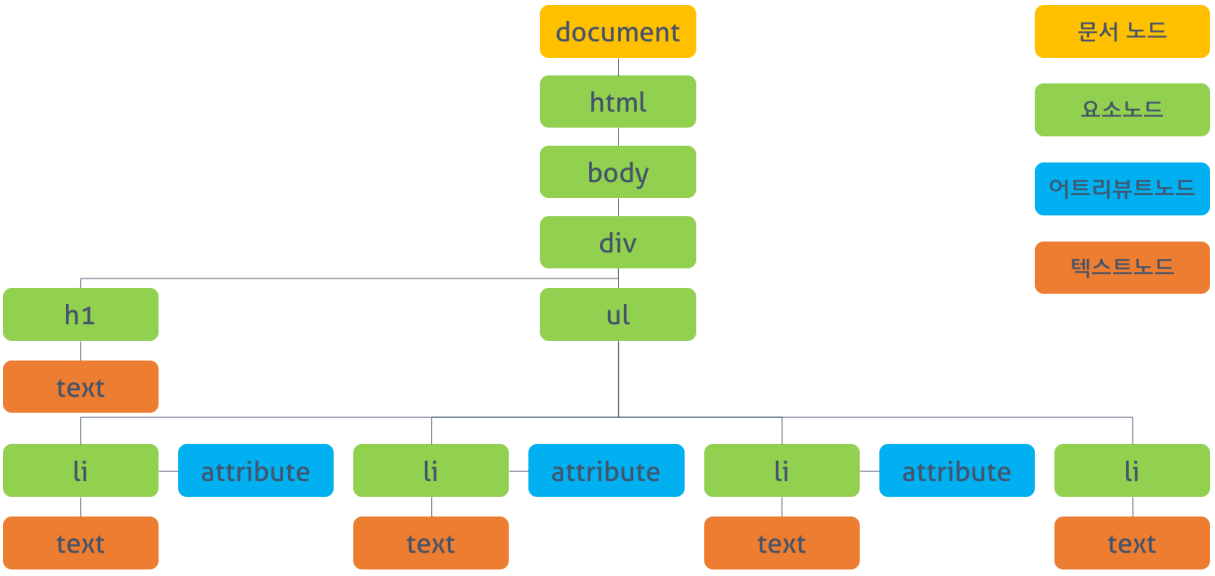
HTML 문서에 대한 모델 구성브라우저는 HTML 문서를 로드한 후 해당 문서에 대한 모델을 메모리에 생성한다. 이때 모델은 객체의 트리로 구성되는데 이것을 **DOM tree**라 한다.**HTML 문서 내의 각 요소에 접근 / 수정**DOM은 모델 내의 각 객체에 접근하고 수정할 수 있는 프로퍼티와 메소드를 제공한다. DOM이 수정되면 브라우저를 통해 사용자가 보게 될 내용 또한 변경된다.

DOM tree

DOM tree는 브라우저가 HTML 문서를 로드한 후 파싱하여 생성하는 모델을 의미한다. 객체의 트리로 구조화되어 있기 때문에 DOM tree라 부른다.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .red  { color: #ff0000; }
      .blue { color: #0000ff; }
    </style>
  </head>
  <body>
    <div>
      <h1>Cities</h1>
      <ul>
        <li id="one" class="red">Seoul</li>
        <li id="two" class="red">London</li>
```

```
        <li id="three" class="red">Newyork</li>
        <li id="four">Tokyo</li>
      </ul>
    </div>
  </body>
</html>
```

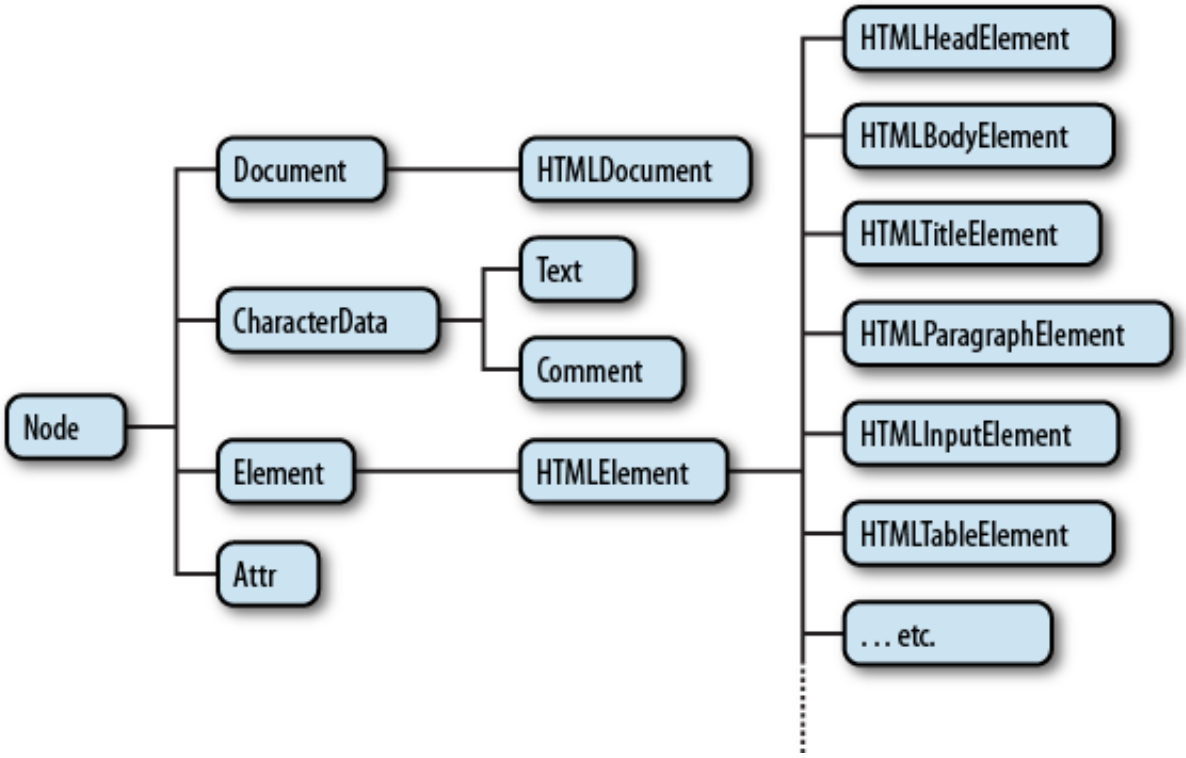


DOM에서 모든 요소, 어트리뷰트, 텍스트는 하나의 객체이며 Document 객체의 자식이다. 요소의 중첩관계는 객체의 트리로 구조화하여 부자관계를 표현한다. DOM tree의 진입점(Entry point)는 document 객체이며 최종점은 요소의 텍스트를 나타내는 객체이다.

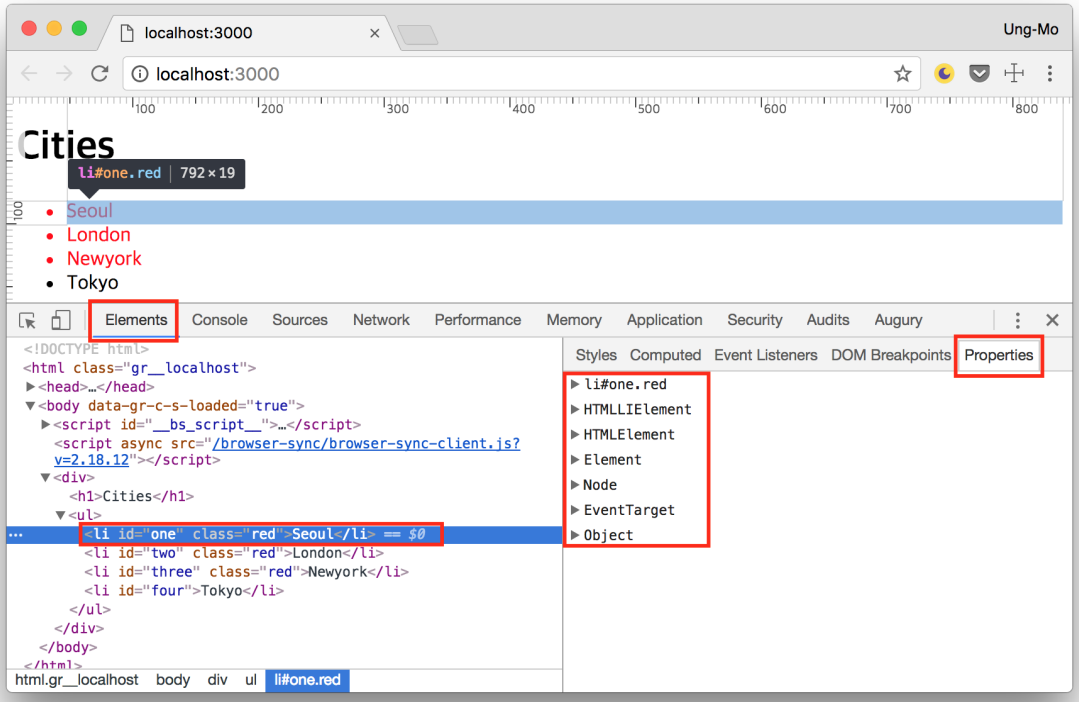
DOM tree는 네 종류의 노드로 구성된다.

문서 노드(Document Node)트리의 최상위에 존재하며 각각 요소, 어트리뷰트, 텍스트 노드에 접근하려면 문서 노드를 통해야 한다. 즉, DOM tree에 접근하기 위한 시작점(entry point)이다.**요소 노드(Element Node)**요소 노드는 HTML 요소를 표현한다. HTML 요소는 중첩에 의해 부자 관계를 가지며 이 부자 관계를 통해 정보를 구조화한다. 따라서 요소 노드는 문서의 구조를 서술한다고 말 할 수 있다. 어트리뷰트, 텍스트 노드에 접근하려면 먼저 요소 노드를 찾아 접근해야 한다. 모든 요소 노드는 요소별 특성을 표현하기 위해 HTMLElement 객체를 상속한 객체로 구성된다. (그림: DOM tree의 객체 구성 참고)**어트리뷰트 노드(Attribute Node)**어트리뷰트 노드는 HTML 요소의 어트리뷰트를 표현한다. 어트리뷰트 노드는 해당 어트리뷰트가 지정된 요소의 자식이 아니라 해당 요소의 일부로 표

현된다. 따라서 해당 요소 노드를 찾아 접근하면 어트리뷰트를 참조, 수정할 수 있다.**텍스트 노드(Text Node)**텍스트 노드는 HTML 요소의 텍스트를 표현한다. 텍스트 노드는 요소 노드의 자식이며 자신의 자식 노드를 가질 수 없다. 즉, 텍스트 노드는 DOM tree의 최종단이다.



DOM 트리를 크롬 브라우저에서 확인하려면 개발자도구(Developer Tools)의 Elements를 선택한 후 오른쪽의 properties을 선택한다.



DOM을 통해 웹페이지를 조작(manipulate)하기 위해서는 다음과 같은 수순이 필요하다.

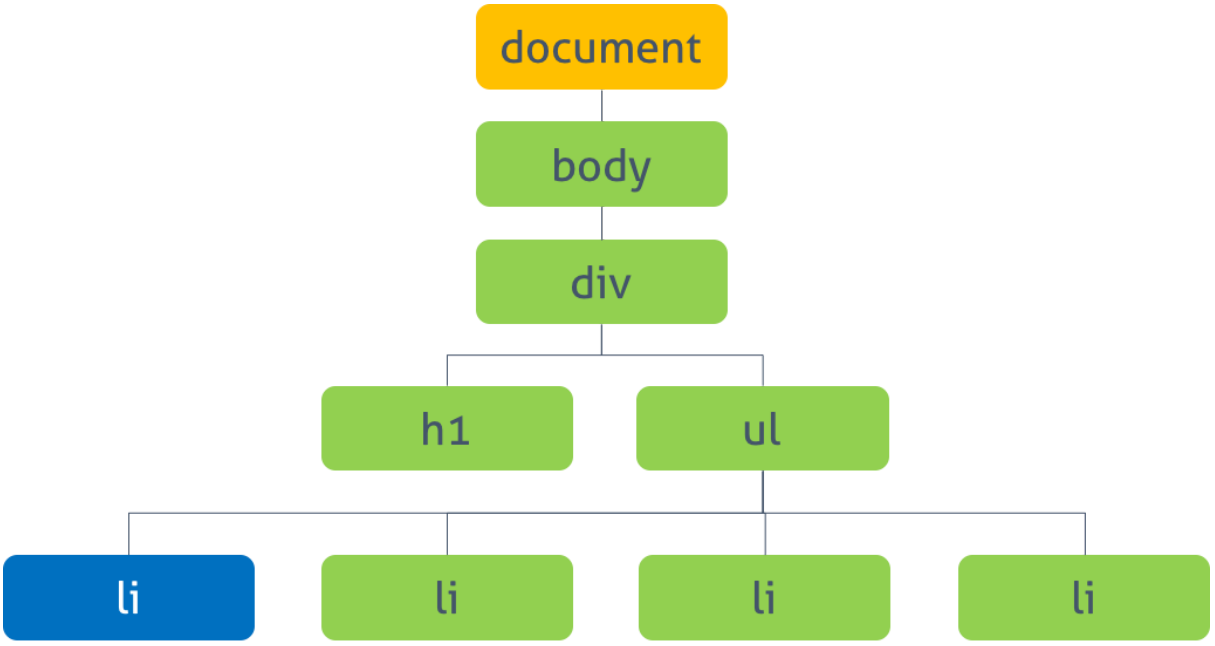
- 조작하고자하는 요소를 선택 또는 탐색한다.

- 선택된 요소의 콘텐츠 또는 어트리뷰트를 조작한다.

자바스크립트는 이것에 필요한 수단(API)을 제공한다.

DOM Query / Traversing (요소에의 접근)

하나의 요소 노드 선택(DOM Query)



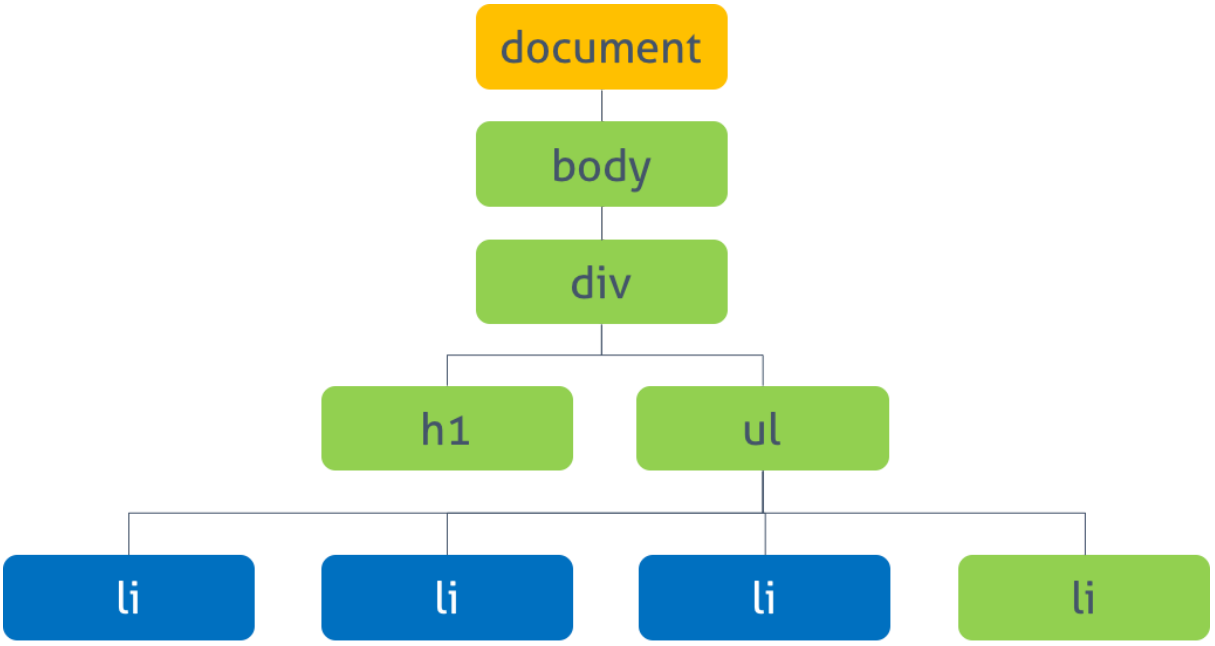
✍🏻 **document.getElementById(id)**
-id 어트리뷰트 값으로 요소 노드를 한 개 선택한다. 복수개가 선택된 경우, 첫번째 요소만 반환한다.
- Return: HTMLElement를 상속받은 객체
- 모든 브라우저에서 동작

```
// id로 하나의 요소를 선택한다.
const elem = document.getElementById('one');
// 클래스 어트리뷰트의 값을 변경한다.
elem.className = 'blue';

// 그림: DOM tree의 객체 구성 참고
console.log(elem); // <li id="one" class="blue">Seoul</li>
console.log(elem.__proto__);           // HTMLLIElement
console.log(elem.__proto__.__proto__); // HTMLElement
console.log(elem.__proto__.__proto__.__proto__);
// Element
console.log(elem.__proto__.__proto__.__proto__.__proto__);
// Node
```

✍🏻 **document.querySelector(cssSelector)**
- CSS 셀렉터를 사용하여 요소 노드를 한 개 선택한다. 복수개가 선택된 경우, 첫 번째 요소만 반환한다.
- Return: HTMLElement를 상속받은 객체
- IE8 이상의 브라우저에서 동작

```
// CSS 셀렉터를 이용해 요소를 선택한다
const elem = document.querySelector('li.red');
// 클래스 어트리뷰트의 값을 변경한다.
elem.className = 'blue';
```

여러 개의 요소 노드 선택(DOM Query)



✍🏻 **document.getElementsByClassName(class)**
- class 어트리뷰트 값으로 요소 노드를 모두 선택한다. 공백으로 구분하여 여러 개의 class를 지정할 수 있다.
- Return: HTMLCollection (live)
- IE9 이상의 브라우저에서 동작

```
// HTMLCollection을 반환한다. HTMLCollection은 live하다.
const elems = document.getElementsByClassName('red');

for (let i = 0; i < elems.length; i++) {
  // 클래스 어트리뷰트의 값을 변경한다.
  elems[i].className = 'blue';
}
```

위 예제를 실행해 보면 예상대로 동작하지 않는다. (두번째 요소만 클래스 변경이 되지 않는다.)

getElementsByClassName 메소드의 반환값은 HTMLCollection이다. 이것은 반환값이 복수인 경우, HTMLElement의 리스트를 담아 반환하기 위한 객체로 배열과 비슷한 사용법을 가지고 있지만 배열은 아닌 **유사배열(array-like object)**이다. 또한 HTMLCollection은 **실시간으로 Node의 상태 변경을 반영한다.** (live HTMLCollection)

위 예제가 예상대로 동작하지 않은 이유를 알아보자.

elems.length는 3이므로 3번의 loop가 실행된다.

1. i가 0일때, elems의 첫 요소(li#one.red)의 class 어트리뷰트의 값이 className 프로퍼티에 의해 red에서 blue로 변경된다. 이때 elems는 실시간으로 Node의 상태 변경을 반영하는 HTMLCollection 객체이다. elems의 첫 요소는 li#one.red에서 li#one.blue로 변경되었으므로 getElementsByClassName 메소드의 인자로 지정한 선택 조건('red')과 더 이상 부합하지 않게 되어 반환값에서 실시간으로 제거된다.

2. i가 1일때, elems에서 첫째 요소는 제거되었으므로 elems[1]은 3번째 요소(li#three.red)가 된다. li#three.red의 class 어트리뷰트 값이 blue로 변경되고 마찬가지로 HTMLCollection에서 제외된다.

3. i가 2일때, HTMLCollection의 1,3번째 요소가 실시간으로 제거되었으므로 2번째 요소(li#two.red)만 남았다. 이때 elems.length는 1이므로 for 문의 조건식 i < elems.length

가 false로 평가되어 반복을 종료한다. 따라서 elems에 남아 있는 2번째 li 요소(li#two.red)의 class 값은 변경되지 않는다.

이처럼 HTMLCollection는 실시간으로 Node의 상태 변경을 반영하기 때문에 loop가 필요한 경우 주의가 필요하다. 아래와 같은 방법으로 회피할 수 있다.

- 반복문을 역방향으로 돌린다.

```
const elems = document.getElementsByClassName('red');

for (let i = elems.length - 1; i >= 0; i--) {
  elems[i].className = 'blue';
}
```

- while 반복문을 사용한다. 이때 elems에 요소가 남아 있지 않을 때까지 무한반복하기 위해 index는 0으로 고정시킨다.

```
const elems = document.getElementsByClassName('red');

let i = 0;
while (elems.length > i) { // elems에 요소가 남아 있지 않을 때까지 무한반복
  elems[i].className = 'blue';
  // i++;
}
```

- HTMLCollection을 배열로 변경한다. 이 방법을 권장한다.

```
const elems = document.getElementsByClassName('red');

// 유사 배열 객체인 HTMLCollection을 배열로 변환한다.
// 배열로 변환된 HTMLCollection은 더 이상 live하지 않다.
console.log([...elems]); // [li#one.red, li#two.red, li#three.red]
```

```
[...elems].forEach(elem => elem.className = 'blue');
```

- querySelectorAll 메소드를 사용하여 HTMLCollection(live)이 아닌 NodeList(non-live)를 반환하게 한다.

```
// querySelectorAll는 Nodelist(non-live)를 반환한다. IE8+
const elems = document.querySelectorAll('.red');

[...elems].forEach(elem => elem.className = 'blue');
```

✍🏻 **document.getElementsByTagName(tagName)**
- 태그명으로 요소 노드를 모두 선택한다.
- Return: HTMLCollection (live)
- 모든 브라우저에서 동작

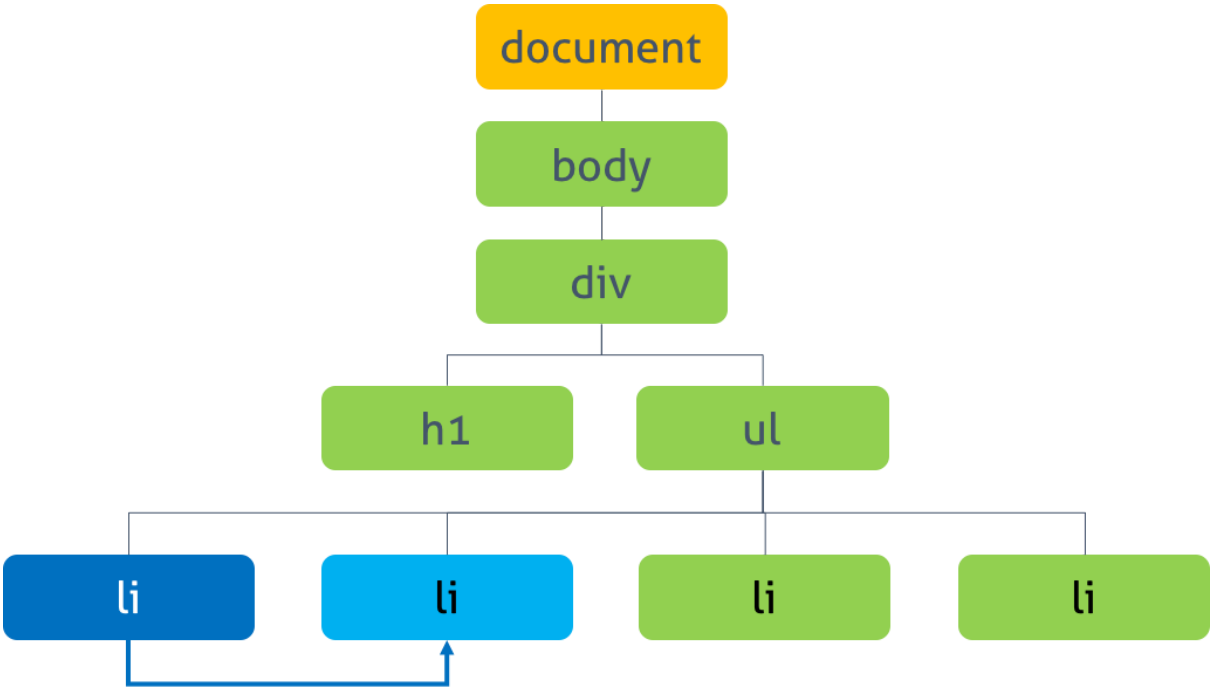
```
// HTMLCollection을 반환한다.
const elems = document.getElementsByTagName('li');

[...elems].forEach(elem => elem.className = 'blue');
```

✍🏻 **document.querySelectorAll(selector)**
- 지정된 CSS 선택자를 사용하여 요소 노드를 모두 선택한다.
- Return: NodeList (non-live)
- IE8 이상의 브라우저에서 동작

```
// Nodelist를 반환한다.
const elems = document.querySelectorAll('li.red');
```

```
[...elems].forEach(elem => elem.className = 'blue');
```

DOM Traversing (탐색)



✍🏻 **parentNode**
- 부모 노드를 탐색한다.
- Return: HTMLElement를 상속받은 객체
- 모든 브라우저에서 동작

```
const elem = document.querySelector('#two');

elem.parentNode.className = 'blue';
```

✍🏻 **firstChild, lastChild**
- 자식 노드를 탐색한다.
- Return: HTMLElement를 상속받은 객체
- IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

// first Child
elem.firstChild.className = 'blue';
// last Child
elem.lastChild.className = 'blue';
```

위 예제를 실행해 보면 예상대로 동작하지 않는다. 그 이유는 IE를 제외한 대부분의 브라우저들은 요소 사이의 공백 또는 줄바꿈 문자를 텍스트 노드로 취급하기 때문이다. 이것을 회피하기 위해서는 아래와 같이 HTML의 공백을 제거하거나 jQuery: .prev()와 jQuery: .next()를 사용한다.

```
<ul><li
  id='one' class='red'>Seoul</li><li
  id='two' class='red'>London</li><li
  id='three' class='red'>Newyork</li><li
  id='four'>Tokyo</li></ul>
```

또는 firstElementChild, lastElementChild를 사용할 수도 있다. 이 두가지 프로퍼티는 모든 IE9 이상에서 정상 동작한다.

```
const elem = document.querySelector('ul');

// first Child
elem.firstElementChild.className = 'blue';
// last Child
elem.lastElementChild.className = 'blue';
```

✍🏻 **hasChildNodes()**
- 자식 노드가 있는지 확인하고 Boolean 값을 반환한다.
- Return: Boolean 값
- 모든 브라우저에서 동작

✍🏻 **childNodes**
- 자식 노드의 컬렉션을 반환한다.
텍스트 요소를 포함한 모든 자식 요소를 반환한다.
- Return: NodeList (non-live)
- 모든 브라우저에서 동작

✍🏻 **children**
- 자식 노드의 컬렉션을 반환한다.
자식 요소 중에서 Element type 요소만을 반환한다.
- Return: HTMLCollection (live)
- IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

if (elem.hasChildNodes()) {
  console.log(elem.childNodes);
  // 텍스트 요소를 포함한 모든 자식 요소를 반환한다.
  // NodeList(9) [text, li#one.red, text, li#two.red, text, li#three.red, text, li#four, text]

  console.log(elem.children);
  // 자식 요소 중에서 Element type 요소만을 반환한다.
  // HTMLCollection(4) [li#one.red, li#two.red, li#three.red, li#four, one: li#one.red, two: li#two.red, three: li#three.red, four: li#four]
  [...elem.children].forEach(el => console.log(el.nodeType)); // 1 (=> Element node)
}
```

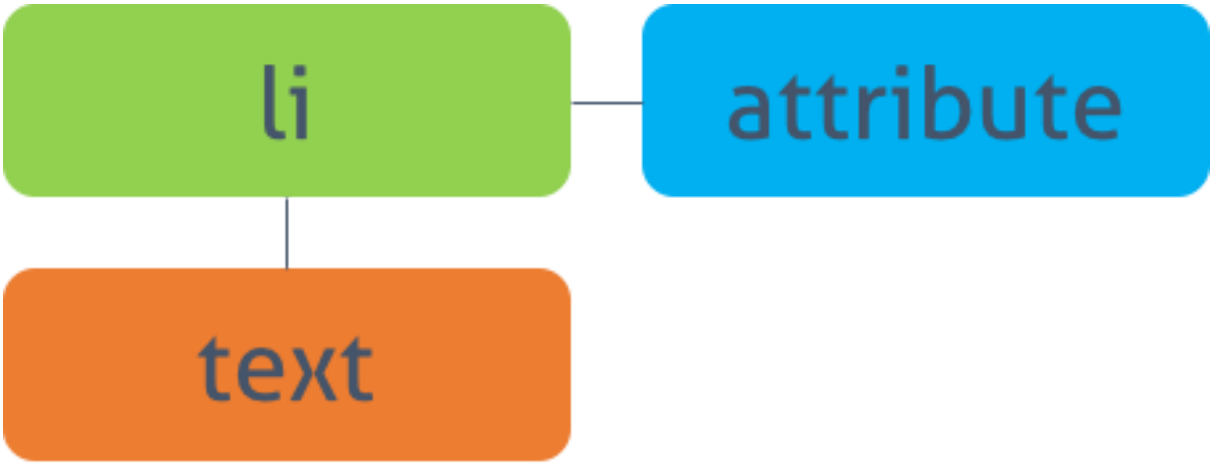
✍🏻 **previousSibling, nextSibling**
- 형제 노드를 탐색한다.
text node를 포함한 모든 형제 노드를 탐색한다.
- Return: HTMLElement를 상속받은 객체
- 모든 브라우저에서 동작

✍🏻 **previousElementSibling, nextElementSibling**
- 형제 노드를 탐색한다.
형제 노드 중에서 Element type 요소만을 탐색한다.
- Return: HTMLElement를 상속받은 객체
- IE9 이상의 브라우저에서 동작

```
const elem = document.querySelector('ul');

elem.firstElementChild.nextElementSibling.className = 'blu
e';
elem.firstElementChild.nextElementSibling.previousElementSi
bling.className = 'blue';
```

DOM Manipulation (조작)

텍스트 노드에의 접근/수정



요소의 텍스트는 텍스트 노드에 저장되어 있다. 텍스트 노드에 접근하려면 아래와 같은 수순이 필요하다.

1. 해당 텍스트 노드의 부모 노드를 선택한다. 텍스트 노드는 요소 노드의 자식이다.
2. firstChild 프로퍼티를 사용하여 텍스트 노드를 탐색한다.
3. 텍스트 노드의 유일한 프로퍼티(`nodeValue`)를 이용하여 텍스트를 취득한다.
4. `nodeValue`를 이용하여 텍스트를 수정한다.

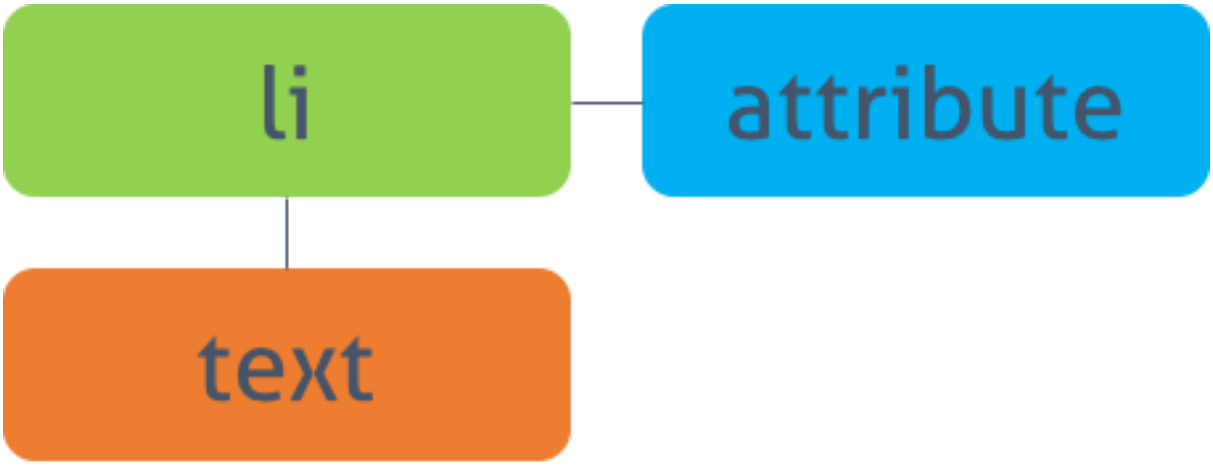
✍🏻 **nodeValue**
- 노드의 값을 반환한다.
- Return: 텍스트 노드의 경우는 문자열, 요소 노드의 경우 null 반환
- IE6 이상의 브라우저에서 동작한다.

nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.

```
// 해당 텍스트 노드의 부모 요소 노드를 선택한다.
const one = document.getElementById('one');
console.dir(one); // HTMLLIElement: li#one.red

// nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.
console.log(one.nodeName); // LI
console.log(one.nodeType); // 1: Element node
```

```
// firstChild 프로퍼티를 사용하여 텍스트 노드를 탐색한다.
const textNode = one.firstChild;

// nodeName, nodeType을 통해 노드의 정보를 취득할 수 있다.
console.log(textNode.nodeName); // #text
console.log(textNode.nodeType); // 3: Text node

// nodeValue 프로퍼티를 사용하여 노드의 값을 취득한다.
console.log(textNode.nodeValue); // Seoul

// nodeValue 프로퍼티를 이용하여 텍스트를 수정한다.
textNode.nodeValue = 'Pusan';
```

어트리뷰트 노드에의 접근/수정



어트리뷰트 노드을 조작할 때 다음 프로퍼티 또는 메소드를 사용할 수 있다.

✍🏻 **className**
- class 어트리뷰트의 값을 취득 또는 변경한다. className 프로퍼티에 값을 할당하는 경우, class 어트리뷰트가 존재하지 않으면 class 어트리뷰트를 생성하고 지정된 값을 설정한다. class 어트리뷰트의 값이 여러 개일 경우, 공백으로 구분된 문자열이 반환되므로 String 메소드 `split(' ')`를 사용하여 배열로 변경하여 사용한다.
- 모든 브라우저에서 동작한다.

✍🏻 **classList**
- add, remove, item, toggle, contains, replace 메소드를 제공한다.
- IE10 이상의 브라우저에서 동작한다.

```
const elems = document.querySelectorAll('li');

// className
[...elems].forEach(elem => {
  // class 어트리뷰트 값을 취득하여 확인
  if (elem.className === 'red') {
    // class 어트리뷰트 값을 변경한다.
    elem.className = 'blue';
  }
});

// classList
[...elems].forEach(elem => {
  // class 어트리뷰트 값 확인
  if (elem.classList.contains('blue')) {
    // class 어트리뷰트 값 변경한다.
    elem.classList.replace('blue', 'red');
  }
});
```

✍🏻 **id**
- id 어트리뷰트의 값을 취득 또는 변경한다. id 프로퍼티에 값을 할당하는 경우, id 어트리뷰트가 존재하지 않으면 id 어트리뷰트를 생성하고 지정된 값을 설정한다.
- 모든 브라우저에서 동작한다.

```
// h1 태그 요소 중 첫번째 요소를 취득
const heading = document.querySelector('h1');

console.dir(heading); // HTMLHeadingElement: h1
console.log(heading.firstChild.nodeValue); // Cities

// id 어트리뷰트의 값을 변경.
// id 어트리뷰트가 존재하지 않으면 id 어트리뷰트를 생성하고 지정된 값을 설정
heading.id = 'heading';
console.log(heading.id); // heading
```

✍🏻 **hasAttribute(attribute)**
- 지정한 어트리뷰트를 가지고 있는지 검사한다.
- Return : Boolean
- IE8 이상의 브라우저에서 동작한다.

✍🏻 **getAttribute(attribute)**
- 어트리뷰트의 값을 취득한다.
- Return : 문자열
- 모든 브라우저에서 동작한다.

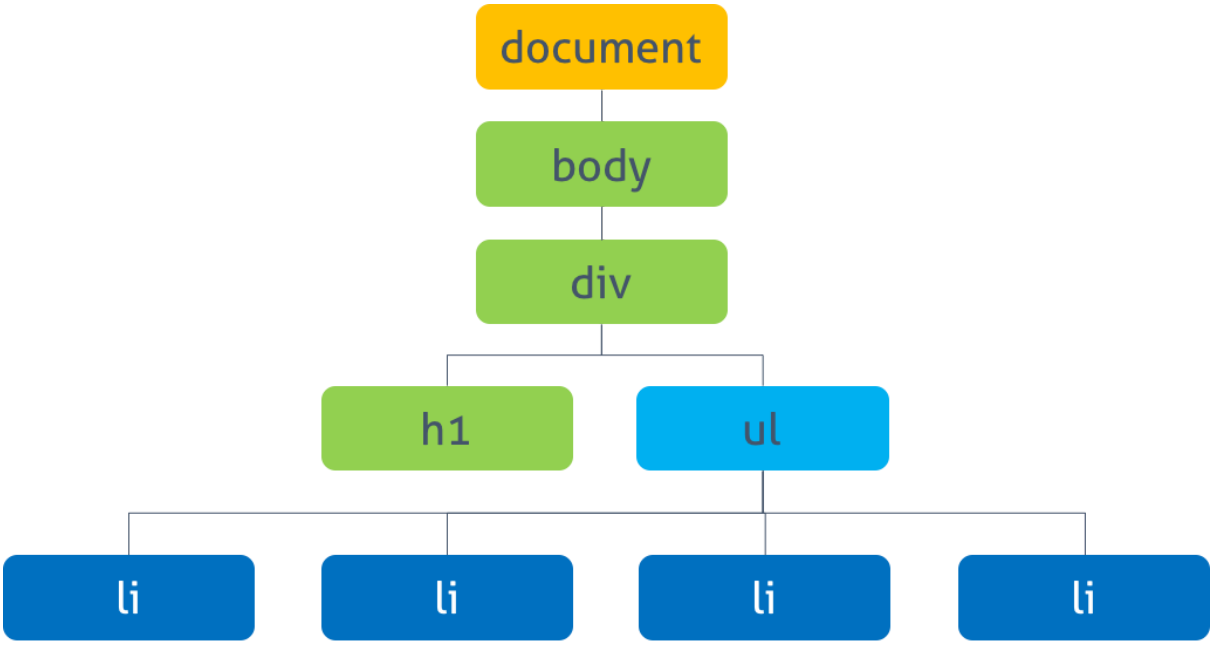
✍🏻 **setAttribute(attribute, value)**
- 어트리뷰트와 어트리뷰트 값을 설정한다.
- Return : undefined
- 모든 브라우저에서 동작한다.

✍🏻 **removeAttribute(attribute)**
- 지정한 어트리뷰트를 제거한다.
- Return : undefined
- 모든 브라우저에서 동작한다.

```
<!DOCTYPE html>
<html>
  <body>
  <input type="text">
  <script>
  const input = document.querySelector('input[type=text]');
  console.log(input);

  // value 어트리뷰트가 존재하지 않으면
  if (!input.hasAttribute('value')) {
    // value 어트리뷰트를 추가하고 값으로 'hello!'를 설정
    input.setAttribute('value', 'hello!');
  }

  // value 어트리뷰트 값을 취득
  console.log(input.getAttribute('value')); // hello!

  // value 어트리뷰트를 제거
  input.removeAttribute('value');

  // value 어트리뷰트의 존재를 확인
  console.log(input.hasAttribute('value')); // false
  </script>
  </body>
</html>
```

```
<!DOCTYPE html>
<html>
<body>
  <input class="password" type="password" value="123">
  <button class="show">show</button>
  <script>
    const $password = document.querySelector('.password');
    const $show = document.querySelector('.show');

    function makeClickHandler() {
      let isShow = false;
      return function () {
        $password.setAttribute('type', isShow ? 'password'
: 'text');
        isShow = !isShow;
        $show.innerHTML = isShow ? 'hide' : 'show';
      };
    }

    $show.onclick = makeClickHandler();
  </script>
</body>
</html>
```

HTML 콘텐츠 조작(Manipulation)



HTML 콘텐츠를 조작(Manipulation)하기 위해 아래의 프로퍼티 또는 메소드를 사용할 수 있다. 마크업이 포함된 콘텐츠를 추가하는 행위는 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하므로 주의가 필요하다.

textContent
- 요소의 텍스트 콘텐츠를 취득 또는 변경한다. 이때 마크업은 무시된다.
textContent를 통해 요소에 새로운 텍스트를 할당하면 텍스트를 변경할 수 있다. 이때 순수한 텍스트만 지정해야 하며 마크업을 포함시키면 문자열로 인식되어 그대로 출력된다.
- IE9 이상의 브라우저에서 동작한다.

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      .red  { color: #ff0000; }
      .blue { color: #0000ff; }
    </style>
  </head>
  <body>
```

```
<div>
  <h1>Cities</h1>
  <ul>
    <li id="one" class="red">Seoul</li>
    <li id="two" class="red">London</li>
    <li id="three" class="red">Newyork</li>
    <li id="four">Tokyo</li>
  </ul>
</div>
<script>
const ul = document.querySelector('ul');

// 요소의 텍스트 취득
console.log(ul.textContent);
/*
        Seoul
        London
        Newyork
        Tokyo
*/

const one = document.getElementById('one');

// 요소의 텍스트 취득
console.log(one.textContent); // Seoul

// 요소의 텍스트 변경
one.textContent += ', Korea';

console.log(one.textContent); // Seoul, Korea

// 요소의 마크업이 포함된 콘텐츠 변경.
one.textContent = '<h1>Heading</h1>';

// 마크업이 문자열로 표시된다.
console.log(one.textContent); // <h1>Heading</h1>
</script>
```

```
  </body>
</html>
```

✍🏻 **innerText**
- innerText 프로퍼티를 사용하여도 요소의 텍스트 콘텐츠에만 접근할 수 있다. 하지만 아래의 이유로 사용하지 않는 것이 좋다.
  - 비표준이다.
  - CSS에 순종적이다. 예를 들어 CSS에 의해 비표시(visibility: hidden;)로 지정되어 있다면 텍스트가 반환되지 않는다.
  - CSS를 고려해야 하므로 textContent 프로퍼티보다 느리다

✍🏻 **innerHTML**
- 해당 요소의 모든 자식 요소를 포함하는 모든 콘텐츠를 하나의 문자열로 취득할 수 있다. 이 문자열은 마크업을 포함한다.

```
const ul = document.querySelector('ul');

// innerHTML 프로퍼티는 모든 자식 요소를 포함하는 모든 콘텐츠를 하나의 문자열로 취득할 수 있다. 이 문자열은 마크업을 포함한다.
console.log(ul.innerHTML);
// IE를 제외한 대부분의 브라우저들은 요소 사이의 공백 또는 줄바꿈 문자를 텍스트 노드로 취급한다
/*
        <li id="one" class="red">Seoul</li>
        <li id="two" class="red">London</li>
        <li id="three" class="red">Newyork</li>
        <li id="four">Tokyo</li>
*/
```

innerHTML 프로퍼티를 사용하여 마크업이 포함된 새로운 콘텐츠를 지정하면 새로운 요소를 DOM에 추가할 수 있다.

```
const one = document.getElementById('one');

// 마크업이 포함된 콘텐츠 취득
console.log(one.innerHTML); // Seoul

// 마크업이 포함된 콘텐츠 변경
one.innerHTML += '<em class="blue">, Korea</em>';

// 마크업이 포함된 콘텐츠 취득
console.log(one.innerHTML); // Seoul <em class="blue">, Kor
ea</em>
```

위와 같이 마크업이 포함된 콘텐츠를 추가하는 것은 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하다.

```
// 크로스 스크립팅 공격 사례

// 스크립트 태그를 추가하여 자바스크립트가 실행되도록 한다.
// HTML5에서 innerHTML로 삽입된 <script> 코드는 실행되지 않는다.
// 크롬, 파이어폭스 등의 브라우저나 최신 브라우저 환경에서는 작동하지 않
을 수도 있다.
elem.innerHTML = '<script>alert("XSS!")</script>';

// 에러 이벤트를 발생시켜 스크립트가 실행되도록 한다.
// 크롬에서도 실행된다!
elem.innerHTML = '<img src="#" onerror="alert(\'XSS\')">';
```

DOM 조작 방식

innerHTML 프로퍼티를 사용하지 않고 새로운 콘텐츠를 추가할 수 있는 방법은 DOM을 직접 조작하는 것이다. 한 개의 요소를 추가하는 경우 사용한다. 이 방법은 다음의 수순에 따라 진행한다.

1. 요소 노드 생성 createElement() 메소드를 사용하여 새로운 요소 노드를 생성한다. createElement() 메소드의 인자로 태그 이름을 전달한다.

2. 텍스트 노드 생성 createTextNode() 메소드를 사용하여 새로운 텍스트 노드를 생성한다. 경우에 따라 생략될 수 있지만 생략하는 경우, 콘텐츠가 비어 있는 요소가 된다.

3. 생성된 요소를 DOM에 추가 appendChild() 메소드를 사용하여 생성된 노드를 DOM tree에 추가한다. 또는 removeChild() 메소드를 사용하여 DOM tree에서 노드를 삭제할 수도 있다.

✍🏻 **createElement(tagName)**
- 태그이름을 인자로 전달하여 요소를 생성한다.
- Return: HTMLElement를 상속받은 객체
- 모든 브라우저에서 동작한다.

✍🏻 **createTextNode(text)**
- 텍스트를 인자로 전달하여 텍스트 노드를 생성한다.
- Return: Text 객체
- 모든 브라우저에서 동작한다.

✍🏻 **appendChild(Node)**
- 인자로 전달한 노드를 마지막 자식 요소로 DOM 트리에 추가한다.
- Return: 추가한 노드
- 모든 브라우저에서 동작한다.

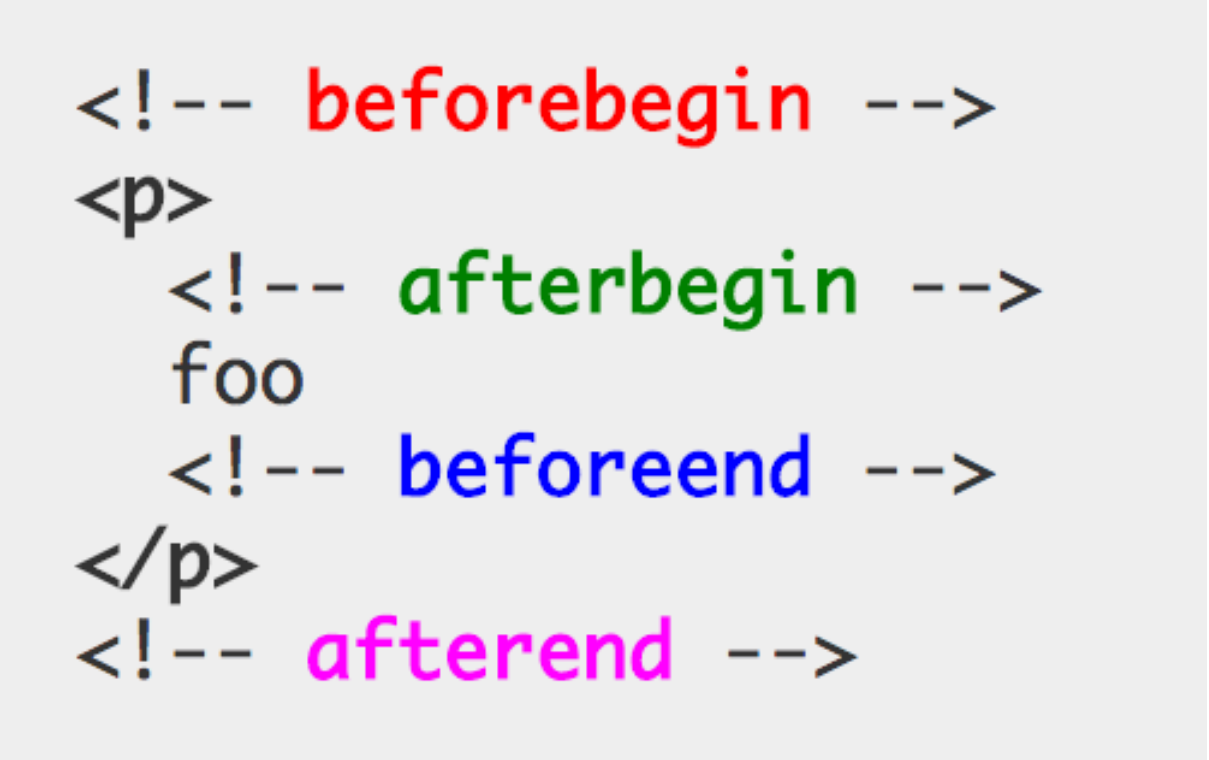
✍🏻 **removeChild(Node)**
- 인자로 전달한 노드를 DOM 트리에 제거한다.
- Return: 추가한 노드
- 모든 브라우저에서 동작한다.

```
// 태그이름을 인자로 전달하여 요소를 생성
const newElem = document.createElement('li');
// const newElem = document.createElement('<li>test</li>');
// Uncaught DOMException: Failed to execute 'createElement'
on 'Document': The tag name provided ('<li>test</li>') is n
ot a valid name.
```

```
// 텍스트 노드를 생성
const newText = document.createTextNode('Beijing');

// 텍스트 노드를 newElem 자식으로 DOM 트리에 추가
newElem.appendChild(newText);

const container = document.querySelector('ul');

// newElem을 container의 자식으로 DOM 트리에 추가. 마지막 요소로 추가된다.
container.appendChild(newElem);

const removeElem = document.getElementById('one');

// container의 자식인 removeElem 요소를 DOM 트리에 제거한다.
container.removeChild(removeElem);
```

insertAdjacentHTML()

✍🏻 **insertAdjacentHTML(position, string)**
- 인자로 전달한 텍스트를 HTML로 파싱하고 그 결과로 생성된 노드를 DOM 트리의 지정된 위치에 삽입한다. 첫번째 인자는 삽입 위치, 두번째 인자는 삽입할 요소를 표현한 문자열이다. 첫번째 인자로 올 수 있는 값은 아래와 같다.
  - 'beforebegin'
  - 'afterbegin'
  - 'beforeend'
  - 'afterend'
- 모든 브라우저에서 동작한다.



insertAdjacentHTML 메소드의 position 파라미터

```
const one = document.getElementById('one');

// 마크업이 포함된 요소 추가
one.insertAdjacentHTML('beforeend', '<em class="blue">, Kor
ea</em>');
```

innerHTML vs. DOM 조작 방식 vs. insertAdjacentHTML()

innerHTML

장점	단점
DOM 조작 방식에 비해 빠르고 간편하다.	XSS공격에 취약점이 있기 때문에 사용자로 부터 입력받은 콘텐츠(untrusted data: 댓글, 사용자 이름 등)를 추가할 때 주의하여야 한다.
간편하게 문자열로 정의한 여러 요소를 DOM에 추가할 수 있다.	해당 요소의 내용을 덮어 쓴다. 즉, HTML을 다시 파싱한다. 이것은 비효율적이다.
콘텐츠를 취득할 수 있다.	

DOM 조작 방식

장점	단점
특정 노드 한 개(노드, 텍스트, 데이터 등)를 DOM에 추가할 때 적합하다.	innerHTML보다 느리고 더 많은 코드가 필요하다.

insertAdjacentHTML()

장점	단점
간편하게 문자열로 정의된 여러 요소를 DOM에 추가할 수 있다.	XSS공격에 취약점이 있기 때문에 사용자로 부터 입력받은 콘텐츠(untrusted data: 댓글, 사용자 이름 등)를 추가할 때 주의하여야 한다.
삽입되는 위치를 선정할 수 있다.	

결론

innerHTML과 insertAdjacentHTML()은 크로스 스크립팅 공격(XSS: Cross-Site Scripting Attacks)에 취약하다. 따라서 untrusted data의 경우, 주의하여야 한다. 텍스트를 추가 또는 변경시에는 textContent, 새로운 요소의 추가 또는 삭제시에는 DOM 조작 방식을 사용하도록 한다.

style

style 프로퍼티를 사용하면 inline 스타일 선언을 생성한다. 특정 요소에 inline 스타일을 지정하는 경우 사용한다.

```
const four = document.getElementById('four');

// inline 스타일 선언을 생성
four.style.color = 'blue';

// font-size와 같이 '-'으로 구분되는 프로퍼티는 카멜케이스로 변환하여 사용한다.
four.style.fontSize = '2em';
```

style 프로퍼티의 값을 취득하려면 window.getComputedStyle을 사용한다. window.getComputedStyle 메소드는 인자로 주어진 요소의 모든 CSS 프로퍼티 값을 반환한다.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initia
l-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>style 프로퍼티 값 취득</title>
  <style>
    .box {
      width: 100px;
      height: 50px;
      background-color: red;
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <div class="box"></div>
  <script>
    const box = document.querySelector('.box');

    const width = getStyle(box, 'width');
    const height = getStyle(box, 'height');
    const backgroundColor = getStyle(box, 'background-colo
r');
    const border = getStyle(box, 'border');

    console.log('width: ' + width);
    console.log('height: ' + height);
    console.log('backgroundColor: ' + backgroundColor);
    console.log('border: ' + border);

    /**
```

```
      * 요소에 적용된 CSS 프로퍼티를 반환한다.
      * @param {HTTPElement} elem - 대상 요소 노드.
      * @param {string} prop - 대상 CSS 프로퍼티.
      * @returns {string} CSS 프로퍼티의 값.
      */
     function getStyle(elem, prop) {
       return window.getComputedStyle(elem, null).getPropert
yValue(prop);
     }
   </script>
</body>
</html>
```