

# DOM과 Event 관계

## DOM 정의

DOM Tree

DOM Tree 노드의 종류

DOM Tree를 크롬브라우저에서 확인하는 방법

## DOM 제어

DOM을 조작하기 위한 순서

하나의 요소 노드 선택

`document.getElementById(id)`

`document.querySelector(cssSelector)`

여러 개의 요소 노드 선택

`document.getElementsByTagName(tagName)`

`document.querySelectorAll(selector)`

## 탐색

`parentNode`

`firstChild, lastChild`

`hasChildNodes()`

`childNodes`

`children`

`previousSibling, nextSibling`

`previousElementSibling, nextElementSibling`

조작 - 텍스트 노드에 접근/수정

`nodeValue`

조작 - 어트리뷰트 노드에의 접근/수정

`className`

classList

id

hasAttribute(attribute)

getAttribute(attribute)

setAttribute(attribute, value)

removeAttribute(attribute)

조작 - HTML 콘텐츠 조작(Manipulation)

textContent

innerText

innerHTML

조작 - DOM 조작 방식

createElement(tagName)

createTextNode(text)

appendChild(node)

removeChild(node)

insertAdjacentHTML()

insertAdjacentHTML(position, string)

innerHTML vs DOM 조작 방식 vs insertAdjacentHTML()

style

과제

자식 DOM

형제 노드에 관한 질문

모든 대각선 셀 선택하기

이벤트 정의

이벤트 종류

UI Event

Keyboard Event

Mouse Event

Focus Event

Form Event

Clipboard Event

이벤트 처리 방법

인라인 이벤트 핸들러 방식

이벤트 핸들러 프로퍼티 방식

addEventListener 메소드 방식

이벤트 핸들러 함수 내부의 this

인라인 이벤트 핸들러 방식

이벤트 핸들러 프로퍼티 방식

addEventListener 메소드 방식

이벤트 위임

버블링

캡처링

버블링 + 캡처링 혼용

Event 객체

Event Property

Event.target

Event.currentTarget

Event.type

Event.cancelable

Event.eventPhase

이벤트 위임 Event Delegation 활용편

li요소를 id로 찾아서 이벤트 위임을 처리하는 방법

data-action 속성을 이용한 이벤트 위임을 처리 방법

이벤트 위임에 이점은 무엇인가?

행동 패턴

토글러 구현하기

요약

장점 - 가독성, 유연성, 최적화

단점

기본 동작 변경

Event.preventDefault()

Event.stopPropagation()

preventDefault & stopPropagation

과제

메시지 숨기기

트리 메뉴 구현하기

정렬 기능을 제공하는 표

툴팁 보여주기

## DOM 정의

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.

DOM = 객체 > 자식객체 ⇒ 트리구조로 정의한 것

객체 = 모든요소 + 어트리뷰트 + 텍스트

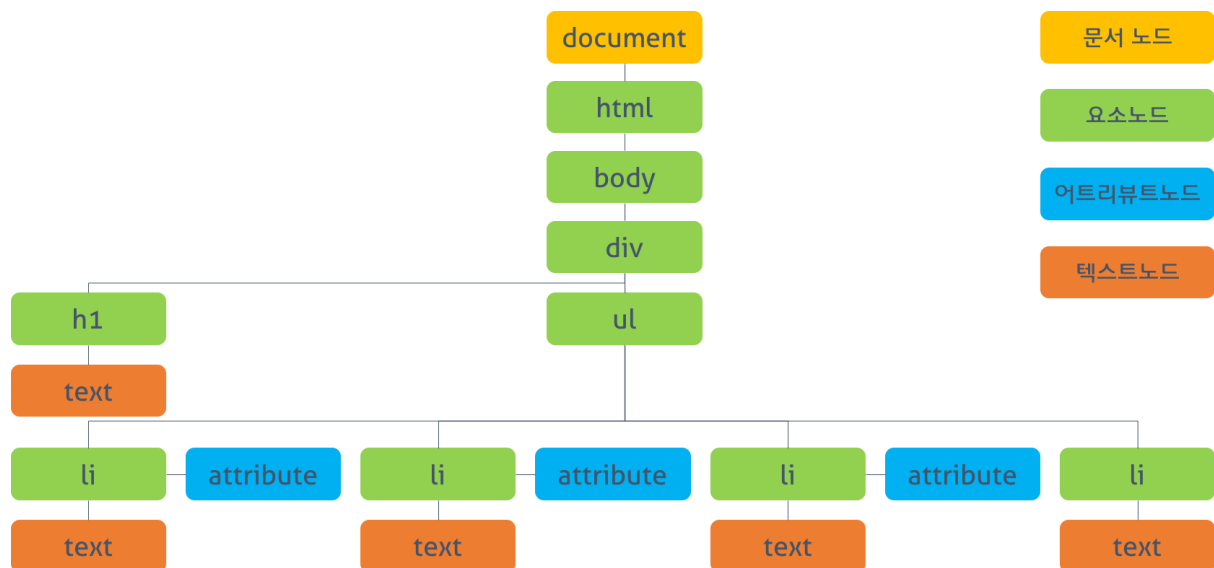
## DOM Tree

참고: <https://ko.javascript.info/dom-nodes>

```

<!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>

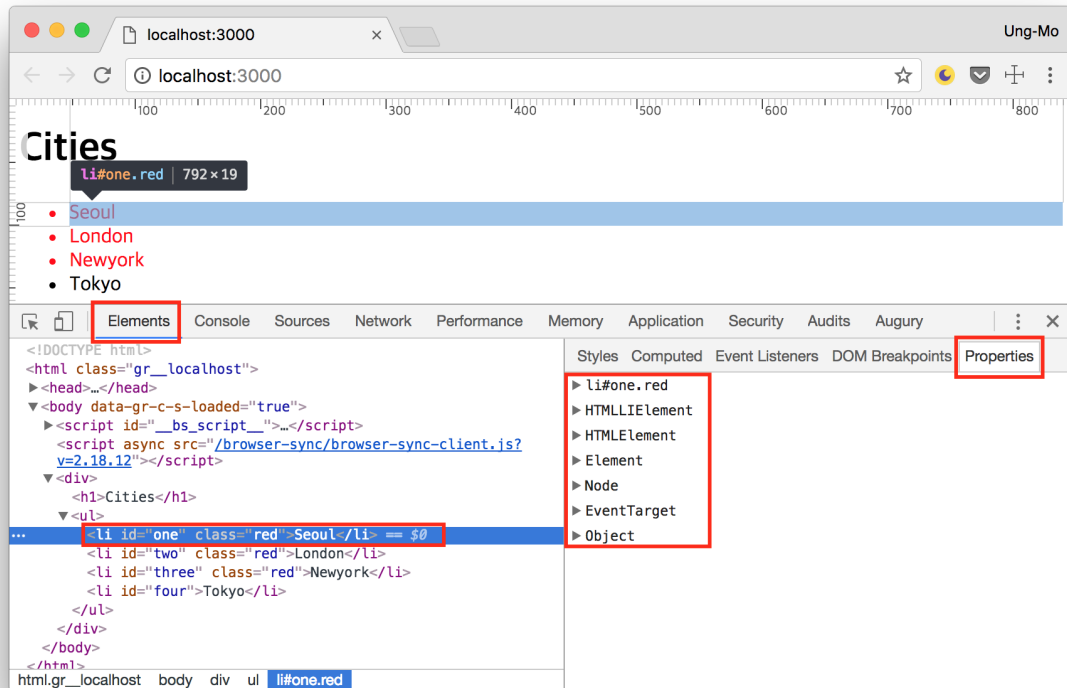
```



## DOM Tree 노드의 종류

- 문서 노드 (Document Node) = DOM Tree 접근하기 위한 시작점
- 요소 노드 (Element Node) = selector
- 어트리뷰트 노드 (Attribute Node) = 해당 요소의 일부로 표현
- 텍스트 노드 (Text Node) = 텍스트 표현

## DOM Tree를 크롬브라우저에서 확인하는 방법



## DOM 제어

웹 문서의 동적 변경을 위해 DOM은 프로그래밍 언어가 자신에 접근하고 수정할 수 있는 방법을 제공한다.

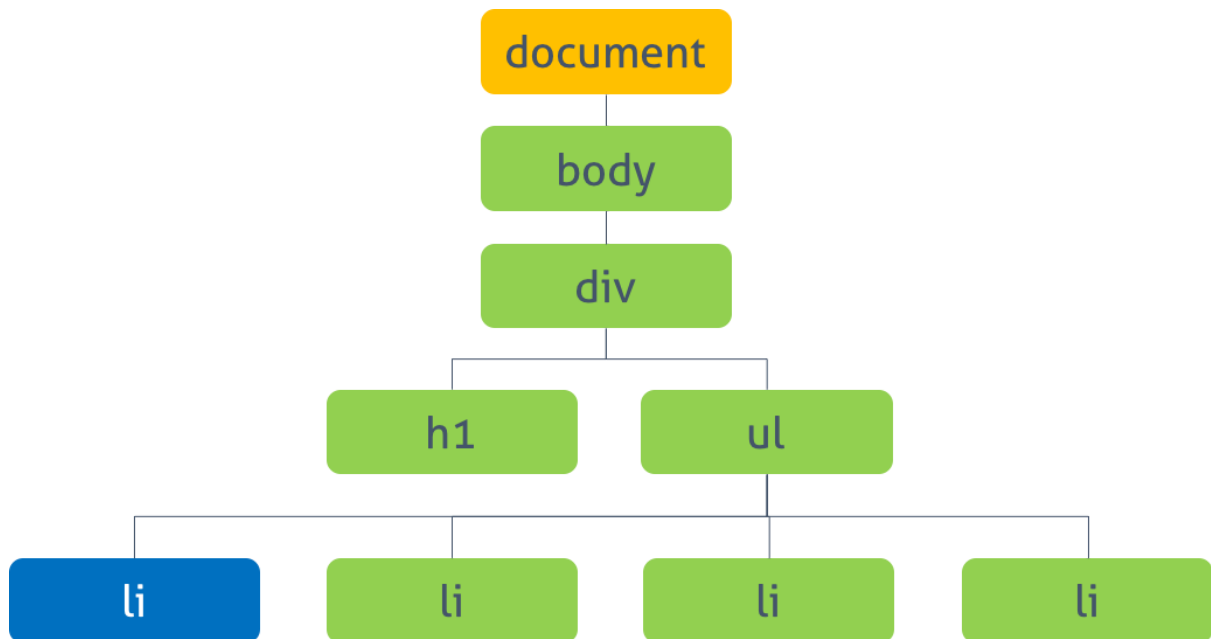
일반적으로 프로퍼티, 메소드를 갖는 자바스크립트 객체로 제공한다.

## DOM을 조작하기 위한 순서

1. 조작하고자하는 요소를 선택 또는 탐색 한다.
2. 선택된 요소의 콘텐츠 또는 어트리뷰트를 조작한다.

자바스크립트는 이것에 필요한 수단 (API)를 제공한다.

## 하나의 요소 노드 선택



## 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';

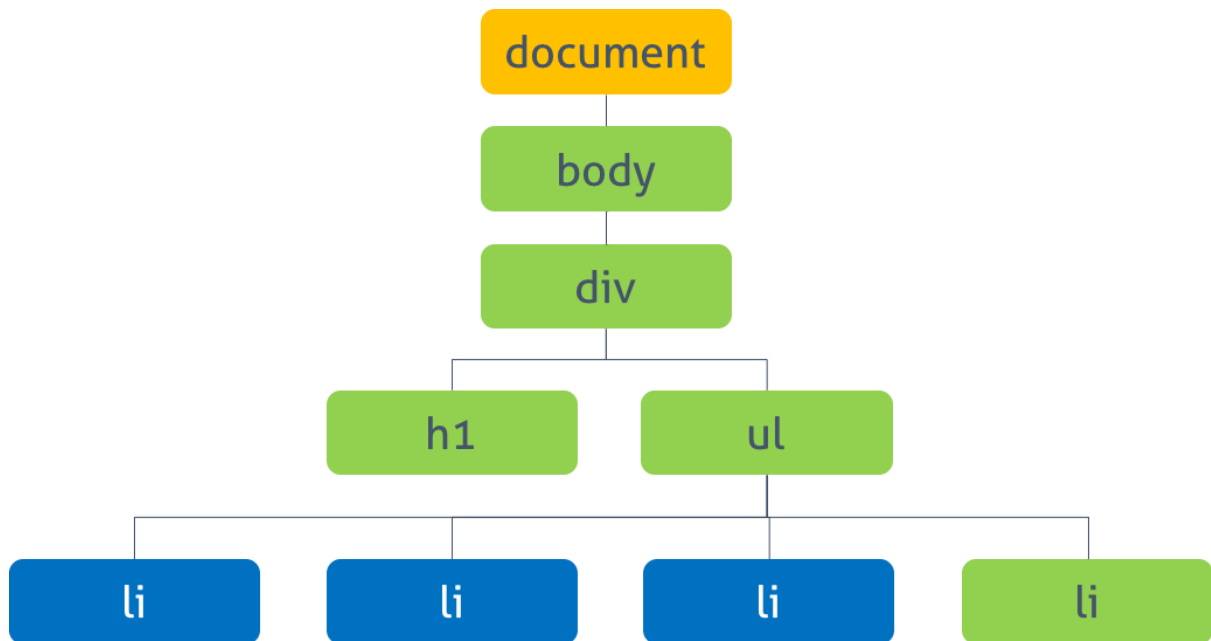
// 그림: DOM tree의 객체 구성 참고
console.log(elem); // <li id="one" class="blue">Seoul</li>
console.log(elem.__proto__); // HTMLLIElement
console.log(elem.__proto__.__proto__); // HTMLElement
console.log(elem.__proto__.__proto__.__proto__); // Element
console.log(elem.__proto__.__proto__.__proto__.__proto__);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

## document.querySelector(cssSelector)

- CSS 셀렉터를 사용하여 요소 노드를 한 개 선택한다. 복수개가 선택된 경우, 첫번째 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



### 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.)

회피 방법, 반복문을 역방향으로 돌린다.

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

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

while 반복문 사용

```
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. 권장 하는 방법

```
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 메소드 사용, 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');
```

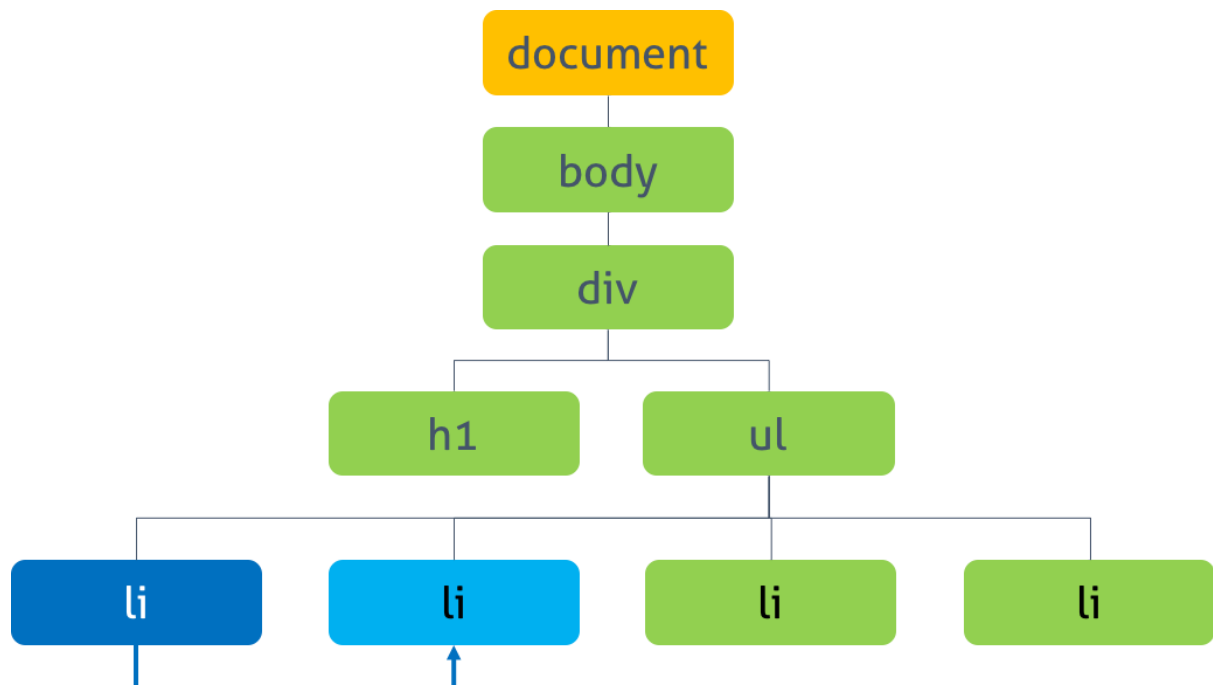
## document.querySelectorAll(selector)

- 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');
```

## 탐색

참고 : <https://ko.javascript.info/dom-navigation>



## 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 예제 정상 동작 안함 ⇒ 요소 사이의 공백 또는 줄바꿈 문자를 텍스트 노드로 취급하기 때문

```
<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>
```

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

// first Child
elem.firstChild.className = 'blue';
// last Child
elem.lastChild.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.childNodes()) {
  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

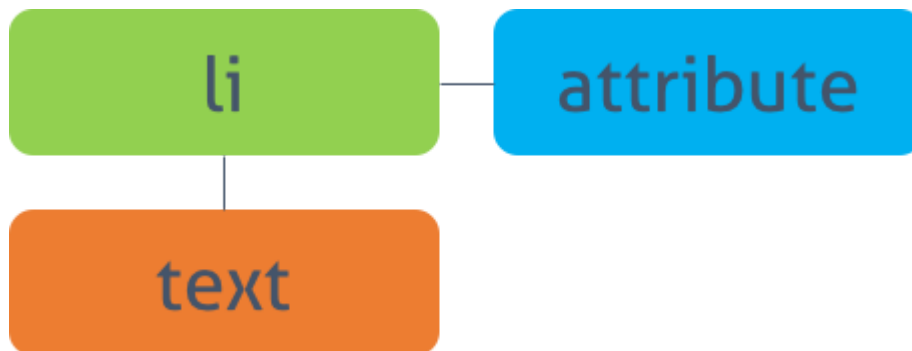
## previousElementSibling, nextElementSibling

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

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

elem.firstChild.nextElementSibling.className = 'blue';
elem.firstChild.nextElementSibling.previousElementSibling.className = 'blue';
```

## 조작 - 텍스트 노드에 접근/수정



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.

```
// 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

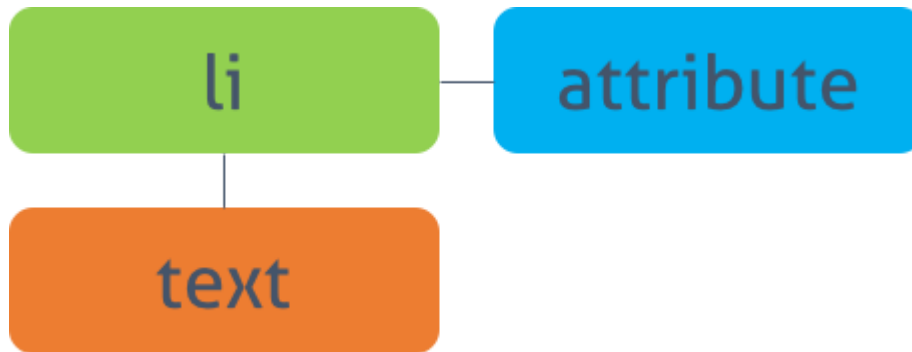
// firstChild 프로퍼티를 사용하여 텍스트 노드를 탐색한다.
const textNode = one.firstChild;

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

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

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

## 조작 - 어트리뷰트 노드와의 접근/수정



### 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';
      };
    }

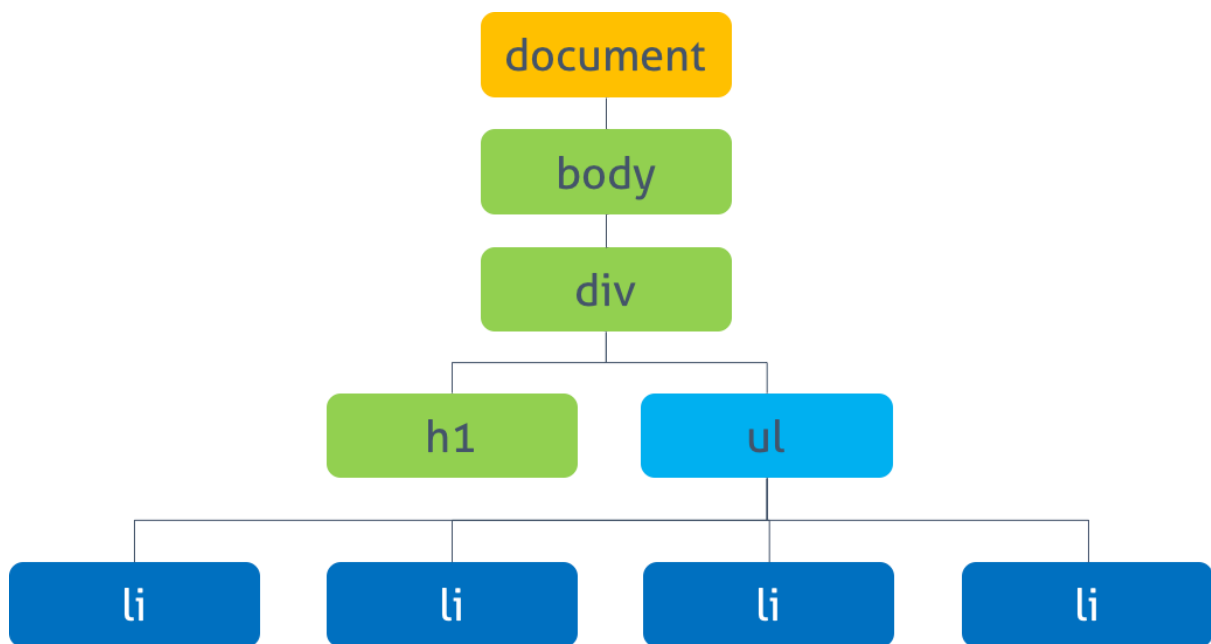
    $show.onclick = makeClickHandler();
```

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

RESULT

... show

## 조작 - 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. `textContent`를 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

## 다.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">, Korea</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 않을 수도 있다.
elem.innerHTML = '<script>alert("XSS!")</script>';

// 예러 이벤트를 발생시켜 스크립트가 실행되도록 한다.
// 크롬에서도 실행된다!
elem.innerHTML = '';
```

## 조작 - 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 not 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.

```

<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->

```

```

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

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

```

## innerHTML vs DOM 조작 방식 vs insertAdjacentHTML()

### innerHTML

장점	단점
DOM 조작 방식에 비해 빠르고 간편하다.	XSS공격에 취약점이 있기 때문에 사용자로부터 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.getComputedStyle`을 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, initial-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-color');
        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 {HTMLElement} elem - 대상 요소 노드.
         * @param {string} prop - 대상 CSS 프로퍼티.
         * @returns {string} CSS 프로퍼티의 값.
         */
        function getStyle(elem, prop) {
            return window.getComputedStyle(elem, null).getPropertyValue(prop);
        }
    </script>
</body>
</html>

```

## 과제

### 자식 DOM

<https://ko.javascript.info/task/dom-children>

### 형제 노드에 관한 질문

<https://ko.javascript.info/task/navigation-links-which-null>

### 모든 대각선 셀 선택하기

<https://ko.javascript.info/task/select-diagonal-cells>

# 이벤트 정의

이벤트(event)는 어떤 사건을 의미한다. 브라우저에서의 이벤트란 예를 들어 사용자가 버튼을 클릭했을 때, 웹페이지가 로드되었을 때와 같은 것인데 이것은 DOM 요소와 관련이 있다.

이벤트가 발생하는 시점이나 순서를 사전에 인지할 수 없으므로 일반적인 제어 흐름과는 다른 접근 방식이 필요하다. 즉, 이벤트가 발생하면 누군가 이를 감지할 수 있어야 하며 그에 대응하는 처리를 호출해 주어야 한다.

브라우저는 이벤트를 감지할 수 있으며 이벤트 발생 시에는 통지해 준다. 이 과정을 통해 사용자와 웹페이지는 상호작용(Interaction)이 가능하게 된다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="myButton">Click me!</button>
  <script>
    document.querySelector('.myButton').addEventListener('click', function () {
      alert('Clicked!');
    });
  </script>
</body>
</html>
```

이벤트가 발생하면 그에 맞는 반응을 하여야 한다. 이를 위해 이벤트는 일반적으로 함수에 연결되며 그 함수는 이벤트가 발생하기 전에는 실행되지 않다가 이벤트가 발생되면 실행된다. 이 함수를 **이벤트 핸들러**라 하며 이벤트에 대응하는 처리를 기술한다.

# 이벤트 종류

## UI Event

Event	Description
<b>load</b>	웹페이지의 로드가 완료되었을 때
unload	웹페이지가 언로드될 때(주로 새로운 페이지를 요청한 경우)
error	브라우저가 자바스크립트 오류를 만났거나 요청한 자원이 존재하지 않는 경우
resize	브라우저 창의 크기를 조절했을 때
scroll	사용자가 페이지를 위아래로 스크롤할 때
select	텍스트를 선택했을 때



## Keyboard Event

Event	Description
keydown	키를 누르고 있을 때
<b>keyup</b>	누르고 있던 키를 떼어낼 때
keypress	키를 누르고 떼어낼 때

## Mouse Event

Event	Description
<b>click</b>	마우스 버튼을 클릭했을 때
dblclick	마우스 버튼을 더블 클릭했을 때
mousedown	마우스 버튼을 누르고 있을 때
mouseup	누르고 있던 마우스 버튼을 떼어낼 때
mousemove	마우스를 움직일 때 (터치스크린에서 동작하지 않는다)
mouseover	마우스를 요소 위로 움직였을 때 (터치스크린에서 동작하지 않는다)
mouseout	마우스를 요소 밖으로 움직였을 때 (터치스크린에서 동작하지 않는다)

## Focus Event

Event	Description
<b>focus</b> /focusin	요소가 포커스를 얻었을 때
<b>blur</b> /focusout	요소가 포커스를 잃었을 때

## Form Event

Event	Description
<b>input</b>	input 또는 textarea 요소의 값이 변경되었을 때
	contenteditable 어트리뷰트를 가진 요소의 값이 변경되었을 때
<b>change</b>	select box, checkbox, radio button의 상태가 변경되었을 때
submit	form을 submit할 때 (버튼 또는 키)
reset	reset 버튼을 클릭할 때 (최근에는 사용 안함)

## Clipboard Event

--	--

Event	Description
cut	콘텐츠를 잘라내기할 때
copy	콘텐츠를 복사할 때
paste	콘텐츠를 붙여넣기할 때

## 이벤트 처리 방법

### 인라인 이벤트 핸들러 방식

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="myHandler()">Click me</button>
  <script>
    function myHandler() {
      alert('Button clicked!');
    }
  </script>
</body>
</html>
```

이 방식은 더 이상 사용되지 않으며 사용해서도 않된다. 오래된 코드에서 간혹 이 방식을 사용한 것이 있기 때문에 알아둘 필요는 있다. HTML과 Javascript는 관심사가 다르므로 분리하는 것이 좋다.



최근 관심을 받고 있는 CBD(Component Based Development) 방식의 Angular/React/Vue.js와 같은 프레임워크/라이브러리에서는 인라인 이벤트 핸들러 방식으로 이벤트를 처리한다. CBD에서는 HTML, CSS, 자바스크립트를 뷰를 구성하기 위한 구성 요소로 보기 때문에 관심사가 다르다고 생각하지 않는다.

주의할 것은 onclick과 같이 on으로 시작하는 이벤트 어트리뷰트의 값으로 함수 호출을 전달한다는 것이다. 다음에 살펴볼 이벤트 핸들러 프로퍼티 방식에는 DOM 요소의 이벤트 핸들러 프로퍼티에 함수 호출이 아닌 함수를 전달한다.

이때 이벤트 어트리뷰트의 값으로 전달한 함수 호출이 즉시 호출되는 것은 아니다. 사실은 이벤트 어트리뷰트 키를 이름으로 갖는 함수를 암묵적으로 정의하고 그 함수의 몸체에 이벤트 어트리뷰트의 값으로 전달한 함수 호출을 문으로 갖는다. 위 예제의 경우, button 요소의 onclick 프로퍼티에 함수 `function onclick(event) { foo(); }`가 할당된다.

즉, 이벤트 어트리뷰트의 값은 암묵적으로 정의되는 이벤트 핸들러의 문이다. 따라서 아래와 같이 여러 개의 문을 전달할 수 있다.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="myHandler1(); myHandler2();">Click me</button>
  <script>
    function myHandler1() {
      alert('myHandler1');
    }
    function myHandler2() {
      alert('myHandler2');
    }
  </script>
</body>
</html>
```

## 이벤트 핸들러 프로퍼티 방식

인라인 이벤트 핸들러 방식처럼 HTML과 Javascript가 뒤섞이는 문제는 해결할 수 있는 방식이다. 하지만 이벤트 핸들러 프로퍼티에 하나의 이벤트 핸들러만을 바인딩할 수 있다는 단점이 있다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="btn">Click me</button>
  <script>
    const btn = document.querySelector('.btn');

    // 이벤트 핸들러 프로퍼티 방식은 이벤트에 하나의 이벤트 핸들러만을 바인딩할 수 있다
    // 첫번째 바인딩된 이벤트 핸들러 => 실행되지 않는다.
    btn.onclick = function () {
      alert('㉠ Button clicked 1');
    };

    // 두번째 바인딩된 이벤트 핸들러
    btn.onclick = function () {
      alert('㉠ Button clicked 2');
    };

    // addEventListener 메소드 방식
    // 첫번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('㉡ Button clicked 1');
    });





    // 두번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('㉡ Button clicked 2');
    });
  </script>
</body>
</html>
```

```
});
</script>
</body>
</html>
```

## addEventListener 메소드 방식

**addEventListener** 메소드를 이용하여 대상 DOM 요소에 이벤트를 바인딩하고 해당 이벤트가 발생했을 때 실행될 콜백 함수(이벤트 핸들러)를 지정한다.

**EventTarget.addEventListener('eventType', functionName [, useCapture]);**

			
대상요소	대상요소에 바인딩될 이벤트를 나타내는 문자열	이벤트 발생 시에 호출될 함수명 또는 함수 자체	capture 사용 여부 true: capturing / false: Bubbling (Default)

addEventListener 함수 방식은 이전 방식에 비해 아래와 같이 보다 나은 장점을 갖는다.

- 하나의 이벤트에 대해 하나 이상의 이벤트 핸들러를 추가할 수 있다.
- 캡처링과 버블링을 지원한다.
- HTML 요소뿐만아니라 모든 DOM 요소(HTML, XML, SVG)에 대해 동작한다. 브라우저는 웹 문서(HTML, XML, SVG)를 로드한 후, 파싱하여 DOM을 생성한다.

addEventListener 메소드는 IE 9 이상에서 동작한다. IE 8 이하에서는 attachEvent 메소드를 사용한다.

```
if (elem.addEventListener) { // IE 9 ~
    elem.addEventListener('click', func);
} else if (elem.attachEvent) { // ~ IE 8
    elem.attachEvent('onclick', func);
}
```

addEventListener 메소드의 사용 예제를 살펴보자.

```
<!DOCTYPE html>
<html>
<body>
  <script>
    addEventListener('click', function () {
      alert('Clicked!');
    });
  </script>
</body>
</html>
```

위와 같이 대상 DOM 요소(target)를 지정하지 않으면 전역객체 window, 즉 DOM 문서를 포함한 브라우저의 윈도우에서 발생하는 click 이벤트에 이벤트 핸들러를 바인딩한다. 따라서 브라우저 윈도우 어디를 클릭하여도 이벤트 핸들러가 동작한다.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>

  <script>
    const input = document.querySelector('input[type=text]');

    input.addEventListener('blur', function () {
      alert('blur event occurred!');
    });
  </script>
</body>
</html>
```

위 예제는 input 요소에서 발생하는 blur 이벤트에 이벤트 핸들러를 바인딩하였다. 사용자 이름이 최소 2자 이상이어야 한다는 규칙을 세우고 이에 부합하는지 확인해보자.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>
  <em class="message"></em>

  <script>
    const input = document.querySelector('input[type=text]');
    const msg = document.querySelector('.message');

    input.addEventListener('blur', function () {
      if (input.value.length < 2) {
        msg.innerHTML = '이름은 2자 이상 입력해 주세요';
      } else {
        msg.innerHTML = '';
      }
    });
  </script>
</body>
</html>
```

2자 이상이라는 규칙이 바뀌면 이 규칙을 확인하는 모든 코드를 수정해야 한다. 따라서 이러한 방식의 코딩은 바람직하지 않다. 이유는 규모가 큰 프로그램의 경우 수정과 테스트에 소요되는 자원의 낭비도 문제이지만 수정에는 거의 대부분 실수가 동반되기 때문이다.

2자 이상이라는 규칙을 상수화하고 함수의 인수로 전달도록 수정하자. 이렇게 하면 규칙이 변경되어도 함수는 수정하지 않아도 된다.

그런데 `addEventListener` 메소드의 두번째 매개변수는 이벤트가 발생했을 때 호출될 이벤트 핸들러이다. 이때 두번째 매개변수에는 함수 호출이 아니라 함수 자체를 지정하여야 한다.

```
function foo() {
  alert('clicked!');
}
// elem.addEventListener('click', foo()); // 이벤트 발생 시까지 대기하지 않고 바로 실행된다
elem.addEventListener('click', foo);      // 이벤트 발생 시까지 대기한다
```

따라서 이벤트 핸들러 프로퍼티 방식과 같이 이벤트 핸들러 함수에 인수를 전달할 수 없는 문제가 발생한다. 이를 우회하는 방법은 아래와 같다.

```
<!DOCTYPE html>
<html>
<body>
  <label>User name <input type='text'></label>
  <em class="message"></em>

  <script>
    const MIN_USER_NAME_LENGTH = 2; // 이름 최소 길이

    const input = document.querySelector('input[type=text]');
    const msg = document.querySelector('.message');

    function checkUserNameLength(n) {
      if (input.value.length < n) {
        msg.innerHTML = '이름은 ' + n + '자 이상이어야 합니다';
      } else {
        msg.innerHTML = '';
      }
    }

    input.addEventListener('blur', function () {
      // 이벤트 핸들러 내부에서 함수를 호출하면서 인수를 전달한다.
      checkUserNameLength(MIN_USER_NAME_LENGTH);
    });

    // 이벤트 핸들러 프로퍼티 방식도 동일한 방식으로 인수를 전달할 수 있다.
    // input.onblur = function () {
    //   // 이벤트 핸들러 내부에서 함수를 호출하면서 인수를 전달한다.
    //   checkUserNameLength(MIN_USER_NAME_LENGTH);
    // };
  </script>
</body>
</html>
```

## 이벤트 핸들러 함수 내부의 this

## 인라인 이벤트 핸들러 방식

인라인 이벤트 핸들러 방식의 경우, 이벤트 핸들러는 일반 함수로서 호출되므로 이벤트 핸들러 내부의 **this**는 전역 객체 **window**를 가리킨다.

```
<!DOCTYPE html>
<html>
<body>
  <button onclick="foo()">Button</button>
  <script>
    function foo () {
      console.log(this); // window
    }
  </script>
</body>
</html>
```

## 이벤트 핸들러 프로퍼티 방식

이벤트 핸들러 프로퍼티 방식에서 이벤트 핸들러는 메소드이므로 이벤트 핸들러 내부의 **this**는 이벤트에 바인딩된 요소를 가리킨다. 이것은 이벤트 객체의 **currentTarget** 프로퍼티와 같다.

```
<!DOCTYPE html>
<html>
<body>
  <button class="btn">Button</button>
  <script>
    const btn = document.querySelector('.btn');

    btn.onclick = function (e) {
      console.log(this); // <button id="btn">Button</button>
      console.log(e.currentTarget); // <button id="btn">Button</button>
      console.log(this === e.currentTarget); // true
    };
  </script>
</body>
</html>
```

## addEventListener 메소드 방식

**addEventListener** 메소드에서 지정한 이벤트 핸들러는 콜백 함수이지만 이벤트 핸들러 내부의 **this**는 이벤트 리스너에 바인딩된 요소(**currentTarget**)를 가리킨다. 이것은 이벤트 객체의 **currentTarget** 프로퍼티와 같다.

```

<!DOCTYPE html>
<html>
<body>
  <button class="btn">Button</button>
  <script>
    const btn = document.querySelector('.btn');

    btn.addEventListener('click', function (e) {
      console.log(this); // <button id="btn">Button</button>
      console.log(e.currentTarget); // <button id="btn">Button</button>
      console.log(this === e.currentTarget); // true
    });
  </script>
</body>
</html>

```

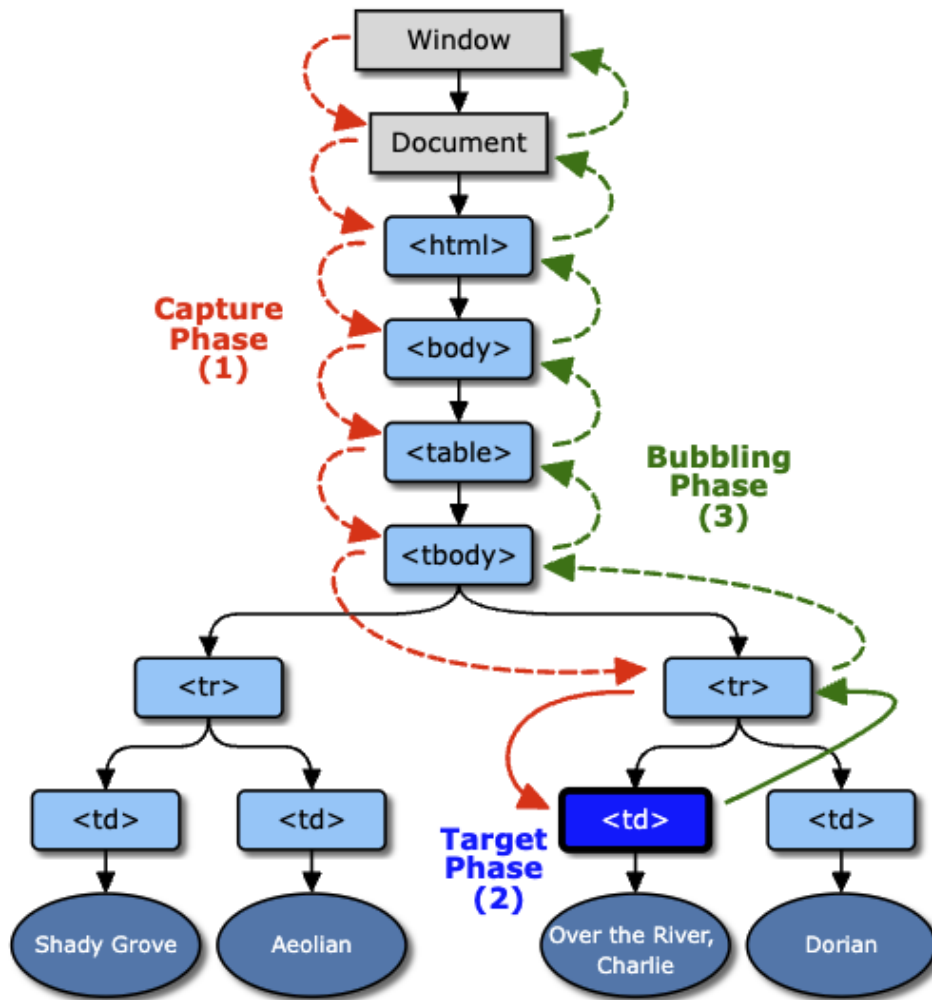
## 이벤트 위임

계층적 구조에 포함되어 있는 HTML 요소에 이벤트가 발생할 경우 연쇄적 반응이 일어난다. 즉, 이벤트가 전파(Event Propagation)되는데 전파 방향에 따라 버블링(Event Bubbling)과 캡처링(Event Capturing)으로 구분할 수 있다.

자식 요소에서 발생한 이벤트가 부모 요소로 전파되는 것을 버블링이라 하고, 자식 요소에서 발생한 이벤트가 부모 요소부터 시작하여 이벤트를 발생시킨 자식 요소까지 도달하는 것을 캡처링이라 한다. **주의할 것은 버블링과 캡처링은 둘 중에 하나만 발생하는 것이 아니라 캡처링부터 시작하여 버블링으로 종료한다는 것이다.** 즉, 이벤트가 발생했을 때 캡처링과 버블링은 순차적으로 발생한다.

캡처링은 IE8 이하에서 지원되지 않는다.





[www.w3.org/TR/DOM-Level-3-Events](http://www.w3.org/TR/DOM-Level-3-Events)

**addEventListener** 메소드의 세번째 매개변수에 **true**를 설정하면 캡처링으로 전파되는 이벤트를 캐치하고 **false** 또는 미설정하면 버블링으로 전파되는 이벤트를 캐치한다.

```
<!DOCTYPE html>
<html>
<head>
<style>
  html { border:1px solid red; padding:30px; text-align: center; }
  body { border:1px solid green; padding:30px; }
  .top {
    width: 300px; height: 300px;
    background-color: red;
    margin: auto;
  }
  .middle {
    width: 200px; height: 200px;
    background-color: blue;
    position: relative; top: 34px; left: 50px;
  }
}
```

```

        .bottom {
            width: 100px; height: 100px;
            background-color: yellow;
            position: relative; top: 34px; left: 50px;
            line-height: 100px;
        }
    </style>
</head>
<body>
    body
    <div class="top">top
        <div class="middle">middle
            <div class="bottom">bottom</div>
        </div>
    </div>
    <script>
    // true: capturing / false: bubbling
    const useCature = true;

    const handler = function (e) {
        const phases = ['capturing', 'target', 'bubbling'];
        const node = this.nodeName + (this.className ? '.' + this.className : '');
        // eventPhase: 이벤트 흐름 상에서 어느 phase에 있는지를 반환한다.
        // 0 : 이벤트 없음 / 1 : 캡처링 단계 / 2 : 타깃 / 3 : 버블링 단계
        console.log(node, phases[e.eventPhase - 1]);
        alert(node + ' : ' + phases[e.eventPhase - 1]);
    };

    document.querySelector('html').addEventListener('click', handler, useCature);
    document.querySelector('body').addEventListener('click', handler, useCature);

    document.querySelector('div.top').addEventListener('click', handler, useCature);
    document.querySelector('div.middle').addEventListener('click', handler, useCature);
    document.querySelector('div.bottom').addEventListener('click', handler, useCature);
    </script>
</body>
</html>

```

## 버블링

```

<!DOCTYPE html>
<html>
<head>
    <style>
        html, body { height: 100%; }
    </style>
<body>
    <p>버블링 이벤트 <button>버튼</button></p>
    <script>
        const body = document.querySelector('body');
        const para = document.querySelector('p');
        const button = document.querySelector('button');

        // 버블링
    </script>

```

```

body.addEventListener('click', function () {
  console.log('Handler for body.');
```

```
});

// 버블링
para.addEventListener('click', function () {
  console.log('Handler for paragraph.');
```

```
});

// 버블링
button.addEventListener('click', function () {
  console.log('Handler for button.');
```

```
});
</script>
</body>
</html>
```

위 코드는 모든 이벤트 핸들러가 이벤트 흐름을 버블링만 캐치한다. 즉, 캡처링 이벤트 흐름에 대해서는 동작하지 않는다. 따라서 button에서 이벤트가 발생하면 모든 이벤트 핸들러는 버블링에 대해 동작하여 아래와 같이 로그된다.

```

Handler for button.
Handler for paragraph.
Handler for body.
```

만약 p 요소에서 이벤트가 발생한다면 p 요소와 body 요소의 이벤트 핸들러는 버블링에 대해 동작하여 아래와 같이 로그된다.

```

Handler for paragraph.
Handler for body.
```

## 캡처링

```

<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
  </style>
<body>
  <p>캡처링 이벤트 <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');
```

```


    // 캡처링
```

```

body.addEventListener('click', function () {
  console.log('Handler for body.');
```

```

}, true);

// 캡처링
para.addEventListener('click', function () {
  console.log('Handler for paragraph.');
```

```

}, true);

// 캡처링
button.addEventListener('click', function () {
  console.log('Handler for button.');
```

```

}, true);
</script>
</body>
</html>

```

위 코드는 모든 이벤트 핸들러가 이벤트 흐름을 캡처링만 캐치한다. 즉, 버블링 이벤트 흐름에 대해서는 동작하지 않는다. 따라서 button에서 이벤트가 발생하면 모든 이벤트 핸들러는 캡처링에 대해 동작하여 아래와 같이 로그된다.

```

Handler for body.
Handler for paragraph.
Handler for button.

```

만약 p 요소에서 이벤트가 발생한다면 p 요소와 body 요소의 이벤트 핸들러는 캡처링에 대해 동작하여 아래와 같이 로그된다.

```

Handler for body.
Handler for paragraph.

```

## 버블링 + 캡처링 혼용

```

<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
  </style>
<body>
  <p>버블링과 캡처링 이벤트 <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');
```

```

    // 버블링

```

```

body.addEventListener('click', function () {
  console.log('Handler for body.');
```

```
});

// 캡처링
para.addEventListener('click', function () {
  console.log('Handler for paragraph.');
```

```
}, true);

// 버블링
button.addEventListener('click', function () {
  console.log('Handler for button.');
```

```
});
</script>
</body>
</html>
```

위 코드의 경우, body, button 요소는 버블링 이벤트 흐름만을 캐치하고 p 요소는 캡처링 이벤트 흐름만을 캐치한다. 따라서 button에서 이벤트가 발생하면 먼저 캡처링이 발생하므로 p 요소의 이벤트 핸들러가 동작하고, 그후 버블링이 발생하여 button, body 요소의 이벤트 핸들러가 동작한다.

```

Handler for paragraph.
Handler for button.
Handler for body.
```

만약 p 요소에서 이벤트가 발생한다면 캡처링에 대해 p 요소의 이벤트 핸들러가 동작하고 버블링에 대해 body 요소의 이벤트 핸들러가 동작한다.

```

Handler for paragraph.
Handler for body.
```

## Event 객체

event 객체는 이벤트를 발생시킨 요소와 발생한 이벤트에 대한 유용한 정보를 제공한다. 이벤트가 발생하면 event 객체는 동적으로 생성되며 이벤트를 처리할 수 있는 이벤트 핸들러에 인자로 전달된다.

```

<!DOCTYPE html>
<html>
<body>
  <p>클릭하세요. 클릭한 곳의 좌표가 표시됩니다.</p>
  <em class="message"></em>
  <script>
    function showCoords(e) { // e: event object
```

```

    const msg = document.querySelector('.message');
    msg.innerHTML =
      'clientX value: ' + e.clientX + '<br>' +
      'clientY value: ' + e.clientY;
  }
  addEventListener('click', showCoords);
</script>
</body>
</html>

```

위와 같이 event 객체는 이벤트 핸들러에 암묵적으로 전달된다. 그러나 이벤트 핸들러를 선언할 때, event 객체를 전달받을 첫번째 매개변수를 명시적으로 선언하여야 한다. 예제에서 e라는 이름으로 매개변수를 지정하였으나 다른 매개변수 이름을 사용하여도 상관없다.

```

<!DOCTYPE html>
<html>
<body>
  <em class="message"></em>
  <script>
    function showCoords(e, msg) {
      msg.innerHTML =
        'clientX value: ' + e.clientX + '<br>' +
        'clientY value: ' + e.clientY;
    }

    const msg = document.querySelector('.message');

    addEventListener('click', function (e) {
      showCoords(e, msg);
    });
  </script>
</body>
</html>

```

## Event Property

### Event.target

실제로 이벤트를 발생시킨 요소를 가리킨다. 아래 예제를 살펴보자.

```

<!DOCTYPE html>
<html>
<body>
  <div class="container">
    <button id="btn1">Hide me 1</button>
    <button id="btn2">Hide me 2</button>
  </div>

```

```

<script>
  function hide(e) {
    e.target.style.visibility = 'hidden';
    // 동일하게 동작한다.
    // this.style.visibility = 'hidden';
  }

  document.getElementById('btn1').addEventListener('click', hide);
  document.getElementById('btn2').addEventListener('click', hide);
</script>
</body>
</html>

```

hide 함수를 특정 노드에 한정하여 사용하지 않고 범용적으로 사용하기 위해 event 객체의 target 프로퍼티를 사용하였다. 위 예제의 경우, hide 함수 내부의 e.target은 언제나 이벤트가 바인딩된 요소를 가리키는 this와 일치한다.

하지만 버튼별로 이벤트를 바인딩하고 있기 때문에 버튼이 많은 경우 위 방법은 바람직하지 않아 보인다.

이벤트 위임을 사용하여 위 예제를 수정하여 보자.

```

<!DOCTYPE html>
<html>
<body>
  <div class="container">
    <button id="btn1">Hide me 1</button>
    <button id="btn2">Hide me 2</button>
  </div>

  <script>
    const container = document.querySelector('.container');

    function hide(e) {
      // e.target은 실제로 이벤트를 발생시킨 DOM 요소를 가리킨다.
      e.target.style.visibility = 'hidden';
      // this는 이벤트에 바인딩된 DOM 요소(.container)를 가리킨다. 따라서 .container 요소를 감춘다.
      // this.style.visibility = 'hidden';
    }

    container.addEventListener('click', hide);
  </script>
</body>
</html>

```

위 예제의 경우, this는 이벤트에 바인딩된 DOM 요소(.container)를 가리킨다. 따라서 container 요소를 감춘다. e.target은 실제로 이벤트를 발생시킨 DOM 요소(button 요소 또는 .container 요소)를 가리킨다. Event.target은 this와 반드시 일치하지는 않는다.

## Event.currentTarget

이벤트에 바인딩된 DOM 요소를 가리킨다. 즉, `addEventListener` 앞에 기술된 객체를 가리킨다.

`addEventListener` 메소드에서 지정한 이벤트 핸들러 내부의 `this`는 이벤트에 바인딩된 DOM 요소를 가리키며 이것은 이벤트 객체의 `currentTarget` 프로퍼티와 같다. 따라서 이벤트 핸들러 함수 내에서 `currentTarget`과 `this`는 언제나 일치한다.

```
<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%; }
    div { height: 100%; }
  </style>
</head>
<body>
  <div>
    <button>배경색 변경</button>
  </div>
  <script>
    function bluify(e) {
      // this: 이벤트에 바인딩된 DOM 요소(div 요소)
      console.log('this: ', this);
      // target: 실제로 이벤트를 발생시킨 요소(button 요소 또는 div 요소)
      console.log('e.target:', e.target);
      // currentTarget: 이벤트에 바인딩된 DOM 요소(div 요소)
      console.log('e.currentTarget: ', e.currentTarget);

      // 언제나 true
      console.log(this === e.currentTarget);
      // currentTarget과 target이 같은 객체일 때 true
      console.log(this === e.target);

      // click 이벤트가 발생하면 이벤트를 발생시킨 요소(target)과는 상관없이 this(이벤트에 바인딩된 div 요소)의 배경색이 변경된다.
      this.style.backgroundColor = '#A5D9F3';
    }

    // div 요소에 이벤트 핸들러가 바인딩되어 있다.
    // 자식 요소인 button이 발생시킨 이벤트가 버블링되어 div 요소에도 전파된다.
    // 따라서 div 요소에 이벤트 핸들러가 바인딩되어 있으면 자식 요소인 button이 발생시킨 이벤트를 div 요소에서도 핸들링할 수 있다.
    document.querySelector('div').addEventListener('click', bluify);
  </script>
</body>
</html>
```

## Event.type

발생한 이벤트의 종류를 나타내는 문자열을 반환한다.



```

<!DOCTYPE html>
<html>
<body>
  <p>키를 입력하세요</p>
  <em class="message"></em>
  <script>
    const body = document.querySelector('body');

    function getEventType(e) {
      console.log(e);
      document.querySelector('.message').innerHTML = `${e.type} : ${e.keyCode}`;
    }

    body.addEventListener('keydown', getEventType);
    body.addEventListener('keyup', getEventType);
  </script>
</body>
</html>

```

## Event.cancelable

요소의 기본 동작을 취소시킬 수 있는지 여부(true/false)를 나타낸다.

```

<!DOCTYPE html>
<html>
<body>
  <a href="poiemaweb.com">Go to poiemaweb.com</a>
  <script>
    const elem = document.querySelector('a');

    elem.addEventListener('click', function (e) {
      console.log(e.cancelable);

      // 기본 동작을 중단시킨다.
      e.preventDefault();
    });
  </script>
</body>
</html>

```

## Event.eventPhase

이벤트 흐름(event flow) 상에서 어느 단계(event phase)에 있는지를 반환한다.

반환값	의미
0	이벤트 없음
1	캡처링 단계

2	타깃
3	버블링 단계

## 이벤트 위임 Event Delegation 활용편

### li요소를 id로 찾아서 이벤트 위임을 처리하는 방법

```
<ul id="post-list">
  <li id="post-1">Item 1</li>
  <li id="post-2">Item 2</li>
  <li id="post-3">Item 3</li>
  <li id="post-4">Item 4</li>
  <li id="post-5">Item 5</li>
  <li id="post-6">Item 6</li>
</ul>

function printId() {
  console.log(this.id);
}

document.querySelector('#post-1').addEventListener('click', printId);
document.querySelector('#post-2').addEventListener('click', printId);
document.querySelector('#post-3').addEventListener('click', printId);
document.querySelector('#post-4').addEventListener('click', printId);
document.querySelector('#post-5').addEventListener('click', printId);
document.querySelector('#post-6').addEventListener('click', printId);
```

만일 li 요소가 100개라면 100개의 이벤트 핸들러를 바인딩하여야 한다. 이는 실행 속도 저하의 원인이 될 뿐 아니라 코드 또한 매우 길어지며 작성 또한 불편하다.

그리고 동적으로 li 요소가 추가되는 경우, 아직 추가되지 않은 요소는 DOM에 존재하지 않으므로 이벤트 핸들러를 바인딩할 수 없다. 이러한 경우 이벤트 위임을 사용한다.

이벤트 위임(Event Delegation)은 다수의 자식 요소에 각각 이벤트 핸들러를 바인딩하는 대신 하나의 부모 요소에 이벤트 핸들러를 바인딩하는 방법이다. 위의 경우 6개의 자식 요소에 각각 이벤트 핸들러를 바인딩하는 것 대신 부모 요소(ul#post-list)에 이벤트 핸들러를 바인딩하는 것이다.

또한 DOM 트리에 새로운 li 요소를 추가하더라도 이벤트 처리는 부모 요소인 ul 요소에 위임되었기 때문에 새로운 요소에 이벤트를 핸들러를 다시 바인딩할 필요가 없다.

이는 이벤트가 이벤트 흐름에 의해 이벤트를 발생시킨 요소의 부모 요소에도 영향(버블링)을 미치기 때문에 가능한 것이다.

실제로 이벤트를 발생시킨 요소를 알아내기 위해서는 `Event.target` 을 사용한다.

```

<!DOCTYPE html>
<html>
<body>
  <ul class="post-list">
    <li id="post-1">Item 1</li>
    <li id="post-2">Item 2</li>
    <li id="post-3">Item 3</li>
    <li id="post-4">Item 4</li>
    <li id="post-5">Item 5</li>
    <li id="post-6">Item 6</li>
  </ul>
  <div class="msg">
  <script>
    const msg = document.querySelector('.msg');
    const list = document.querySelector('.post-list')

    list.addEventListener('click', function (e) {
      // 이벤트를 발생시킨 요소
      console.log('[target]: ' + e.target);
      // 이벤트를 발생시킨 요소의 nodeName
      console.log('[target.nodeName]: ' + e.target.nodeName);

      // li 요소 이외의 요소에서 발생한 이벤트는 대응하지 않는다.
      if (e.target && e.target.nodeName === 'LI') {
        msg.innerHTML = 'li#' + e.target.id + ' was clicked!';
      }
    });
  </script>
</body>
</html>

```

## data-action 속성을 이용한 이벤트 위임을 처리 방법

```

<div id="menu">
  <button data-action="save">저장하기</button>
  <button data-action="load">불러오기</button>
  <button data-action="search">검색하기</button>
</div>

<script>
  class Menu {
    constructor(elem) {
      this._elem = elem;
      elem.onclick = this.onClick.bind(this); // (*)
    }

    save() {
      alert('저장하기');
    }

    load() {
      alert('불러오기');
    }
  }

```

```

search() {
  alert('검색하기');
}

onClick(event) {
  let action = event.target.dataset.action;
  if (action) {
    this[action]();
  }
};
}

new Menu(menu);
</script>

```

## 이벤트 위임에 이점은 무엇인가?

- 버튼마다 핸들러를 할당해주는 코드를 작성할 필요가 없어집니다. 메서드를 만들고 HTML에 그 메서드를 써주기만 하면 됩니다.
- 언제든지 버튼을 추가하고 제거할 수 있어 HTML 구조가 유연해집니다.

`.action-save`, `.action-load` 같은 클래스를 사용할 수도 있지만, `data-action` 속성이 좀 더 의미론적으로 낫습니다. CSS 규칙을 적용할 수도 있게 됩니다.

## 행동 패턴

이벤트 위임은 요소에 *선언적 방식으로* '행동(behavior)'을 추가할 때 사용할 수도 있습니다. 이때는 특별한 속성과 클래스를 사용합니다.

행동 패턴은 두 부분으로 구성됩니다.

1. 요소의 행동을 설명하는 커스텀 속성을 요소에 추가합니다.
2. 문서 전체를 감지하는 핸들러가 이벤트를 추적하게 합니다. 1에서 추가한 속성이 있는 요소에서 이벤트가 발생하면 작업을 수행합니다.

카운터 구현하기

```

첫 번째 카운터: <input type="button" value="1" data-counter>
두 번째 카운터: <input type="button" value="2" data-counter>

<script>
  document.addEventListener('click', function(event) {

    if (event.target.dataset.counter != undefined) { // 속성이 존재할 경우
      event.target.value++;
    }
  })

```

```
});  
</script>
```

버튼을 클릭하면 숫자가 증가합니다. 이 예시에서 집중해야 할 것은 버튼이 아니고 **접근방식**입니다.

`data-counter` 속성이 있는 요소는 원하는 만큼 만들 수 있습니다. 필요할 때마다 HTML에 추가해주면 되니까요. 예시에선 이벤트 위임을 사용해 새로운 행동을 선언해주는 속성을 추가해서 HTML을 '확장'하였습니다.

## 토글러 구현하기

```
<button data-toggle-id="subscribe-mail">  
  구독 폼 보여주기  
</button>  
  
<form id="subscribe-mail" hidden>  
  메일 주소: <input type="email">  
</form>  
  
<script>  
  document.addEventListener('click', function(event) {  
    let id = event.target.dataset.toggleId;  
    if (!id) return;  
  
    let elem = document.getElementById(id);  
  
    elem.hidden = !elem.hidden;  
  });  
</script>
```

자바스크립트를 사용하지 않고도 요소에 토글 기능을 추가할 수 있다는 점에 주목합니다. 태그에 `data-toggle-id` 속성만 추가하면 요소를 토글할 수 있습니다.

행동 패턴을 응용하면 토글 기능이 필요한 요소 전체에 자바스크립트로 해 해당 기능을 구현해 주지 않아도 되기 때문에 매우 편리합니다. '행동'을 선언해 주기만 하면 되기 때문입니다. 문서 레벨에 적절한 핸들러를 구현해주기만 하면 페이지 내 모든 요소에 행동을 쉽게 적용할 수 있습니다.

한 요소에 여러 개의 행동을 조합해 적용하는 것도 가능합니다.

이런 '행동' 패턴은 자바스크립트 미니 프래그먼트의 대안이 될 수 있습니다.

## 요약

이벤트 위임은 상당히 멋진 패턴입니다. DOM 이벤트에 적용할 수 있는 아주 유용한 패턴이죠,

이벤트 위임은 유사한 요소에 동일한 핸들러를 적용할 때 주로 사용하지만 꼭 이런 경우에만 사용할 수 있는 것은 아닙니다.

이벤트 위임은 다음과 같은 알고리즘으로 동작합니다.

1. 컨테이너에 하나의 핸들러를 할당합니다.
2. 핸들러의 `event.target` 을 사용해 이벤트가 발생한 요소가 어디인지 알아냅니다.
3. 원하는 요소에서 이벤트가 발생했다고 확인되면 이벤트를 핸들링합니다.

## 장점 - 가독성, 유연성, 최적화

- 많은 핸들러를 할당하지 않아도 되기 때문에 초기화가 단순해지고 **메모리가 절약**됩니다.
- 요소를 추가하거나 제거할 때 해당 요소에 할당된 핸들러를 추가하거나 제거할 필요가 없기 때문에 **코드가 짧아**집니다.
- `innerHTML` 이나 유사한 기능을 하는 스크립트로 요소 덩어리를 더하거나 뺄 수 있기 때문에 **DOM 수정이 쉬워**집니다.

## 단점

- 이벤트 위임을 사용하려면 이벤트가 반드시 버블링 되어야 합니다. 하지만 몇몇 이벤트는 버블링 되지 않습니다. 그리고 낮은 레벨에 할당한 핸들러엔 `event.stopPropagation()` 를 쓸 수 없습니다.
- 컨테이너 수준에 할당된 핸들러가 응답할 필요가 있는 이벤트이든 아니든 상관없이 모든 하위 컨테이너에서 발생하는 이벤트에 응답해야 하므로 CPU 작업 부하가 늘어날 수 있습니다. 그런데 이런 부하는 무시할만한 수준이므로 실제로는 잘 고려하지 않습니다.

## 기본 동작 변경

이벤트 객체는 요소의 기본 동작과 요소의 부모 요소들이 이벤트에 대응하는 방법을 변경하기 위한 메소드는 가지고 있다.

### Event.preventDefault()

폼을 submit하거나 링크를 클릭하면 다른 페이지로 이동하게 된다. 이와 같이 요소가 가지고 있는 기본 동작을 중단시키기 위한 메소드가 `preventDefault()`이다.

```
<!DOCTYPE html>
<html>
```

```

<body>
  <a href="http://www.google.com">go</a>
  <script>
    document.querySelector('a').addEventListener('click', function (e) {
      console.log(e.target, e.target.nodeName);

      // a 요소의 기본 동작을 중단한다.
      e.preventDefault();
    });
  </script>
</body>
</html>

```

## Event.stopPropagation()

어느 한 요소를 이용하여 이벤트를 처리한 후 이벤트가 부모 요소로 이벤트가 전파되는 것을 중단시키기 위한 메소드이다. 부모 요소에 동일한 이벤트에 대한 다른 핸들러가 지정되어 있을 경우 사용된다.

아래 코드를 보면, 부모 요소와 자식 요소에 모두 mousedown 이벤트에 대한 핸들러가 지정되어 있다. 하지만 부모 요소와 자식 요소의 이벤트를 각각 별도로 처리하기 위해 button 요소의 이벤트의 전파(버블링)를 중단시키기 위해서는 stopPropagation 메소드를 사용하여 이벤트 전파를 중단할 필요가 있다.

```

<!DOCTYPE html>
<html>
<head>
  <style>
    html, body { height: 100%;}
  </style>
</head>
<body>
  <p>버튼을 클릭하면 이벤트 전파를 중단한다. <button>버튼</button></p>
  <script>
    const body = document.querySelector('body');
    const para = document.querySelector('p');
    const button = document.querySelector('button');

    // 버블링
    body.addEventListener('click', function () {
      console.log('Handler for body.');
```

```
    });
```

```
    // 버블링
```

```
    para.addEventListener('click', function () {
      console.log('Handler for paragraph.');
```

```
    });
```

```
    // 버블링
```

```
    button.addEventListener('click', function (event) {
      console.log('Handler for button.');
```

```

        // 이벤트 전파를 중단한다.
        event.stopPropagation();
    });
</script>
</body>
</html>

```

## preventDefault & stopPropagation

기본 동작의 중단과 버블링 또는 캡처링의 중단을 동시에 실시하는 방법은 아래와 같다.

```
return false;
```

단 이 방법은 jQuery를 사용할 때와 아래와 같이 사용할 때만 적용된다.

```

<!DOCTYPE html>
<html>
<body>
  <a href="http://www.google.com" onclick='return handleEvent()'>go</a>
  <script>
    function handleEvent() {
      return false;
    }
  </script>
</body>
</html>

```

```

<!DOCTYPE html>
<html>
<body>
  <div>
    <a href="http://www.google.com">go</a>
  </div>
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.3/jquery.min.js"></script>
  <script>

    // within jQuery
    $('a').click(function (e) {
      e.preventDefault(); // OK
    });

    $('a').click(function () {
      return false; // OK --> e.preventDefault() & e.stopPropagation().
    });

    // pure js
    document.querySelector('a').addEventListener('click', function(e) {

```



```
// e.preventDefault(); // OK
return false;          // NG!!!!
});
</script>
</body>
</html>
```

이 방법은 기본 동작의 중단과 이벤트 흐름의 중단 모두 적용되므로 이 두가지 중 하나만 중단하기 원하는 경우는 `preventDefault()` 또는 `stopPropagation()` 메소드를 개별적으로 사용한다.

## 과제

### 메시지 숨기기

<https://ko.javascript.info/task/hide-message-delegate>

### 트리 메뉴 구현하기

<https://ko.javascript.info/task/sliding-tree>

### 정렬 기능을 제공하는 표

<https://ko.javascript.info/task/sortable-table>

### 툴팁 보여주기

<https://ko.javascript.info/task/behavior-tooltip>