Javascript Event

자료	Javascrtip
구분	Javascript
과목	

Event

오늘은 Javascript에서 “이벤트”에 대해서 공부할 것이다.

이벤트 중 가장 기본적인 클릭 이벤트부터 살펴보자.

어떠한 버튼을 클릭 했을 때 특정 작업이 수행되는 코드를 작성해 볼 것이다.

- 실습해보자.
  1. 아래 버튼을 누르면 숫자가 증가하도록 해보겠다.


```
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0
  <title>Document</title>
 </head>

 <body>
  <button id="btn">버튼</button>
  <p id="counter">0</p>

  <script>
   let cnt = 0;
```

```
    const btn = document.querySelector("#btn");

    btn.addEventListener("click", function () {
      console.log("클릭했음");

      cnt += 1;

      const counter = document.querySelector("#counter");

      counter.innerText = cnt;
    });
  </script>
 </body>

</html>
```

프로그래밍 하고 있는 시스템에서 일어나는 사건을 말한다.

- ex. 클릭 이벤트 실행 → cnt의 값 1 증가

이벤트를 다루는 기본적인 메서드는 addEventListener 이다.

Event를 일으키는 방법은 아주 간단하다.

```
btn.addEventListener('click',function() {
    console.log('클릭했음')
```

- 타겟, 이벤트 그리고 할일(콜백함수)을 가지고 addEventListener() 함수를 이용
- **타겟.addEventListener(이벤트, 할 일)**

여기서 한 가지 짚고 넘어갈 부분이 있다.

아래 코드를 보면 cnt의 값을 1씩 증가 시킨 후

querySelector를 이용해서 `<p id="counter">0</p>` 이 구문을 선택 후

innerText를 이용해서 증가된 값을 <p>태그에 붙였다.

```
btn.addEventListener("click", function () {
  console.log("클릭했음");

  cnt += 1;

  const counter = document.querySelector("#counter");

  counter.innerText = cnt;
```

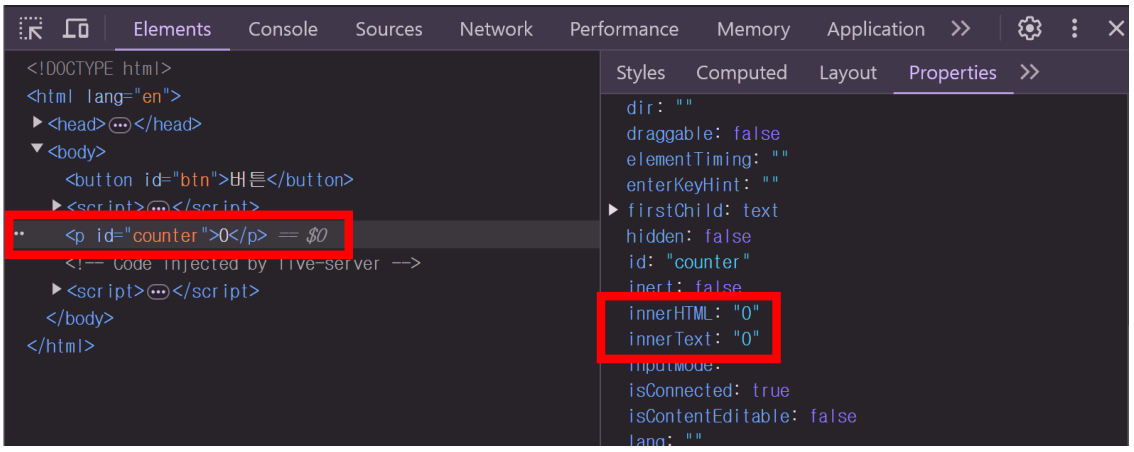
사실 증가 시킨 cnt의 값을 <p> 태그에 붙이는 방법은 여러가지가 있다.

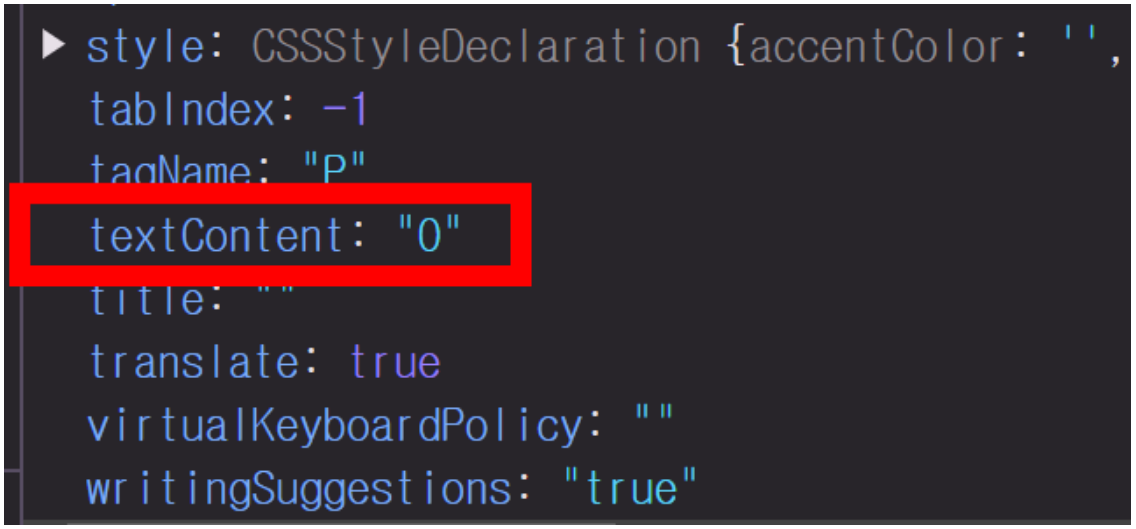
1. innerText를 사용하는 방법
2. innerHTML을 사용하는 방법
3. textContent를 사용하는 방법

코드로 예를 들자면

```
counter.innerText = cnt;
counter.innerHTML = cnt;
connter.textContent = cnt;
```

위 세 가지의 코드는 모두 같은 값을 출력한다.





그래서 우리는 많이 헷갈리는 innerHTML, textContent 그리고 innerText의 차이점에 대해서 확인해 보도록 하자.

1. innerHTML

```
btn.addEventListener("click", function () {
  console.log("클릭했음");

  cnt += 1;

  const counter = document.querySelector("#counter");

  counter.innerHTML = '카운트: <strong>' + cnt + '</strong>';
```

- HTML 태그를 포함 시켜서 HTML요소의 컨텐츠를 가져올 때 사용한다.
- innerHTML은 HTML 태그를 포함할 수 있어 복잡한 콘텐츠를 쉽게 추가할 수 있지만 XSS 공격에 취약하다. 그 이유는 사용자 입력을 직접 설정할 경우 악성 스크립트가 삽입될 수 있기 때문이다.
- 또한 DOM을 재구성해야 하므로 성능이 떨어진다.
- 따라서 content뿐만 아니라 HTML 구조를 같이 변화 시킬 때 사용하면 좋지만 보안 그리고 성능에 취약하다는 점은 기억해 두자.

2. textContent

```
  btn.addEventListener("click", function () {
    console.log("클릭했음");

    cnt += 1;

    const counter = document.querySelector("#counter");

    counter.textContent = '카운트: ' + cnt;
  });
```

- HTML요소의 텍스트만 가져오거나 설정할 때 사용한다.
- HTML 태그를 무시하므로 보안 그리고 성능 상 뛰어나서 HTML 태그를 포함할 필요 없이 content의 내용만 바꿀 때 자주 사용하는 속성이다.
- 텍스트만 바꿀 것이라면 강추다.

3. innerText

```
counter.innerText = '카운트: ' + cnt;
```

- display:none 또는 visibility: hidden과 같은 CSS스타일을 적용하는 경우

  사용자에게 보여지는 텍스트만 가져올 때 사용한다.

예시>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
</head>

<body>

  <div id="example">
```

```
    <span style="display: none;">숨기고 싶은 텍스트</span>
    표시하고자 하는 텍스트
  </div>

  <script>
    const test = document.getElementById('example');
    console.log(test.textContent); // "숨겨진 텍스트 표시된 텍스트"
    console.log(test.innerText);    // "표시된 텍스트"
  </script>
</body>

</html>
```

- textContent를 사용하면 숨긴 요소까지 모두 출력이 되지만
- innerText를 사용하면 숨긴 요소는 가져오지 않고 사용자에게 보여지는 텍스트만 가져온다. 따라서 숨긴 요소가 있다면 필요에 따라 innerText를 사용하면 되겠다.
- 하지만 innerText는 textContent보다 성능이 떨어진다. 그 이유는 innerText가 업데이트 될 때 마다 브라우저는 다시 랜더링 과정을 거치기 때문이다. 하지만 textContent는 단순히 텍스트만 가져와서 처리를 하기 때문에 랜더링을 다시 할 필요가 없기 때문이다.

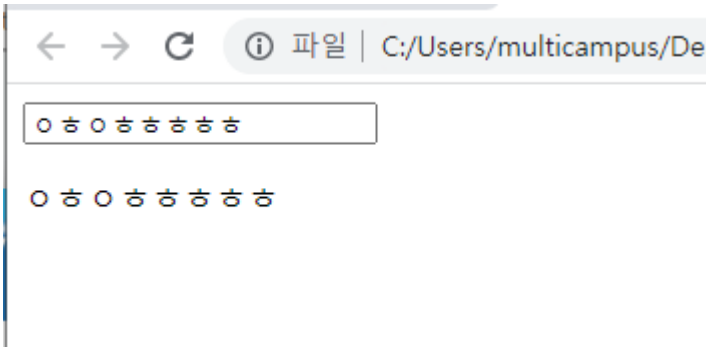
정리를 하자면 다음과 같다.

- **innerHTML**: HTML 구조를 동적으로 변경할 때 유용하지만, 보안과 성능 이슈가 있다.
- **textContent**: 텍스트만 다룰 때 가장 안전하고 빠르지만, HTML 태그를 사용할 수 없다.
- **innerText**: 사용자에게 보여지는 텍스트를 다루는 데 적합하지만, textContent보다 성능이 떨어질 수 있다.

다시 본론으로 돌아오자.

이벤트의 종류는 여러가지가 있다.

이번에는 사용자의 값을 실시간으로 출력하는 input 이벤트를 체험해보자.




```
<body>
  <input type="text" id="text-input">
  <p></p>
  <script>
    //1. input 선택
    const textInput = document.querySelector('#text-input')

    //2. input 이벤트 등록
    textInput.addEventListener('input', function (event) {

      console.log(event.target)        //input은 이벤트의 대상이다.

      console.log(event.target.value)   //input의 value 받아오기

      //3. input에 작성한 값을 p태그에 출력하기
      const pTag = document.querySelector('p')
      pTag.textContent = event.target.value
    })
  </script>
</body>
```

여기에서도 한가지만 짚고 넘어가자.

우리는 이벤트의 대상을 pick 하기 위해서 target 속성을 사용했다.

하지만 target도 가능하지만 아래 그림과 같이

currruntTarget 속성도 사용이 가능하다.

```
<body>
  <input type="text" id="text-input">
  <p></p>
  <script>
    //1. input 선택
    const textInput = document.querySelector('#text-input')

    //2. input 이벤트 등록
    textInput.addEventListener('input', function (event) {

      console.log(event.target)          //input은 이벤트의 대상이다.

      console.log(event.currentTarget.value)   //input의 value 받아오기

      //3. input에 작성한 값을 p태그에 출력하기
      const pTag = document.querySelector('p')
      pTag.textContent = event.currentTarget.value
    })
  </script>
</body>
```

currentTarget 그리고 target은 이벤트 객체에서 사용되는 중요한 속성이다.

하지만 이 둘의 차이점을 알고 사용하는 것 또한 중요하다.

이 둘의 차이점은 뒤에서 이벤트 버블링 이라는 것을 학습 한 후에 살펴 보도록 하자. 궁금하겠지만 조금만 기다려 보자.

이번에는 input에 입력하면 실시간으로 출력하고

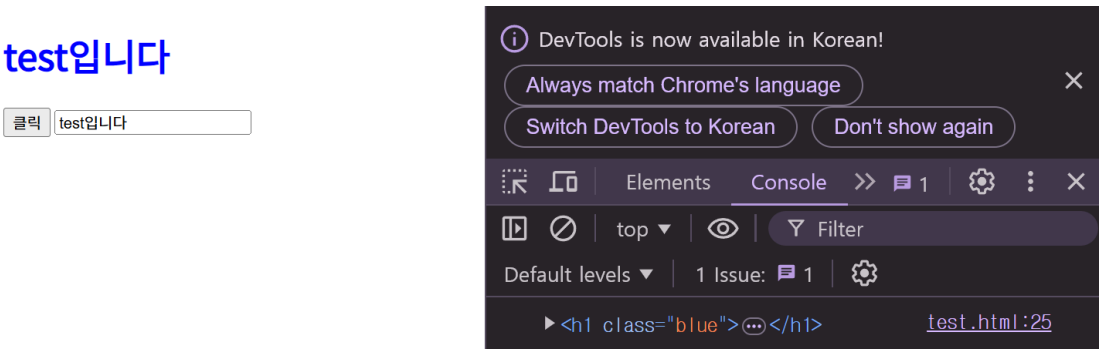
버튼 클릭 시 출력 된 값이 클래스 토글이 되어

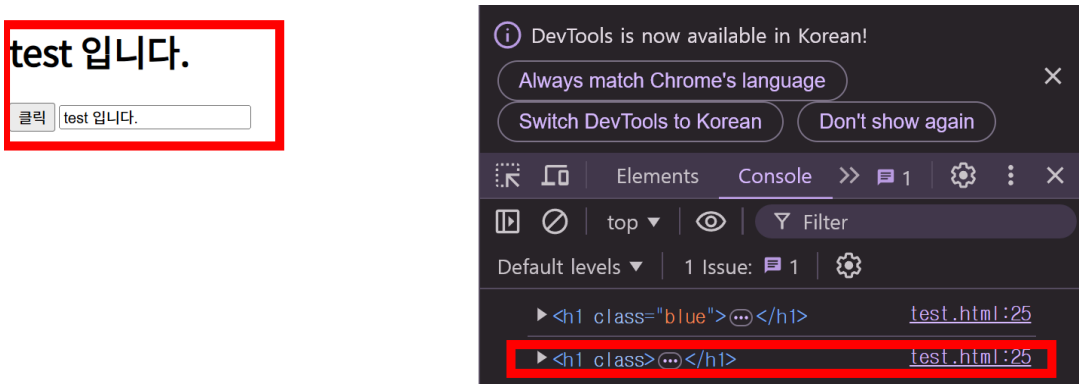
입력한 text가 파란색으로 스타일이 적용이 되도록 해보자.

```
<head>중략
<style>
  .blue {
    color: blue;
  }
 </style>
</head>
<body>
 <h1></h1>
 <button id="btn">클릭</button>
```

```
  <input type="text">

  <script>
   const btn = document.querySelector('#btn')
   btn.addEventListener('click',function() {
    const h1 = document.querySelector('h1')
    h1.classList.toggle('blue')
   })

   const input = document.querySelector('input')
   input.addEventListener('input', function(event) {
    const h1Tag = document.querySelector('h1')
    h1Tag.textContent = event.target.value
   })
  </script>
</body>
```

- classList.toggle : 클래스가 존재하면 제거, false 반환. 존재하지 않으면 추가, true 반환 한다.
  → 위의 코드에서는 원래 지정된 clas가 없으므로 blue 클래스를 추가하고 true를 반환한다.





Event취소

이번에는 이벤트를 취소 시키는 것을 해보자.

이벤트 취소는 **event.preventDefault()** 메서드를 사용해서

현재 Event 기본 동작 중단 할 때 사용한다.

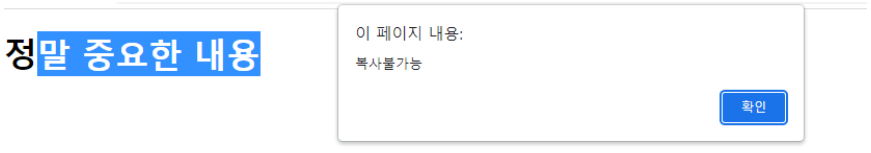
실습해보자.

우리는 복사를 할 때 ctrl + c 를 이용해서 text를 복사를 한다.

하지만 정말 중요한 내용이라서 브라우저에서 복사를 못하게 막고 싶을 때가 있다.

그때 **event.preventDefault()** 메서드를 사용해서 copy라는 이벤트를 막을 수 있다.

```
<body>
  <div>
    <h1>정말 중요한 내용</h1>
  </div>

  <script>
    const h1 = document.querySelector('h1')
    h1.addEventListener('copy', function(event) {
      event.preventDefault()
      alert('복사불가능')
    })
  </script>
</body>
```



정말 중요한 내용을 드래그 이후 copy를 해보자.

alert 창이 뜨면서 복사 불가능 하다는 메시지를 확인 할 수 있을 것이다.

예시>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
</head>

<body>
  <form id="myForm" action='#'>
    <input type="text" class="inputData" required>
    <button type="submit">제출</button>
  </form>

  <script>
    const form = document.getElementById('myForm');

    form.addEventListener('submit', function (event) {
      event.preventDefault();
      // 기본 제출 동작 방지

      // 입력된 값 출력
      const inputData = document.querySelector('.inputData').value;
      console.log("입력된 값: ", inputData);

      // 입력값 유효성검사
      if (!inputData.trim()) {
        alert('입력값이 필요합니다.');
      } else {
        console.log("유효한 데이터: 폼이 제출되었습니다.");

      }
    });
```

```
  </script>
</body>

</html>
```

HTML의 BODY에는 form 태그를 하나 만들어 놓고 사용자로 부터 값을 입력받도록 했다.

form.addEventListener('submit', function (event) {...}) 을 통해서

폼의 submit 이벤트를 감지하고, 사용자가 제출 버튼을 클릭했을 때 실행될 함수를 등록했다.

event.preventDefault(); 를 통해서 submit 버튼을 누르는 이벤트가 실행이 되면

바로 URL로 값이 넘어가지 못하게 이벤트를 막고 유효성 검사를 하는 코드를 작성하였다.

즉, 사용자가 제출 버튼을 클릭하면 기본 제출 동작을 방지하고, 입력값을 검증하여 유효한지 확인하는 코드를 추가 했다.

코드를 추가한 내용은 입력값이 비어 있다면 (if (!inputData.trim()))

경고 메시지를 표시하고, 유효한 경우 콘솔에 메시지를 출력하는 코드이다.

여기까지 event.preventDefault(); 를 보았다. 차후에 비동기 통신에서 자주 사용하는 메서드로 잘 기억을 하고 있으면 좋을 것 같다.

지금까지 학습한 내용을 바탕으로 두가지 실습을 진행 할 것이다.

첫번째는 로또번호를 랜덤하게 생성을 하는 실습

두번째는 클릭을 통해서 todoList를 만드는 실습이다.

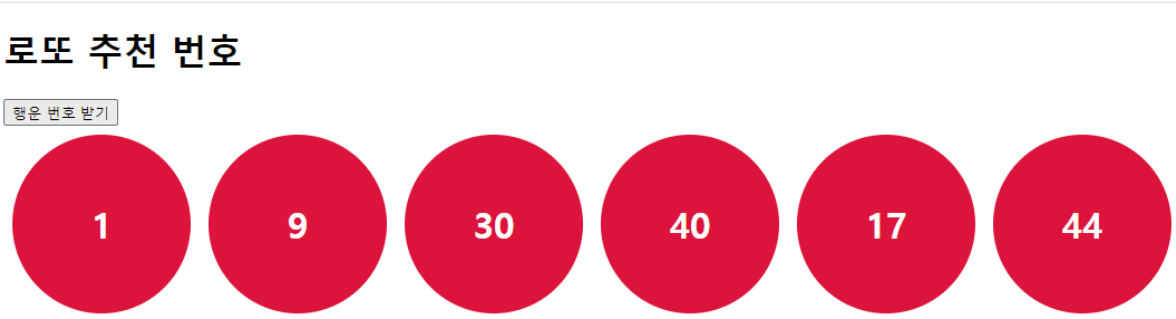
아래 코드들을 직접 복붙 또는 타이핑을 해서 어떤식으로 진행 했는지

눈으로 살펴보고, 브라우저 통해서 확인도 해보자.

1. 로또번호 생성

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
  <style>
    /* 스타일은 수정하지 않습니다. */
    .ball {
      width: 10rem;
      height: 10rem;
      margin: .5rem;
      border-radius: 50%;
      text-align: center;
      line-height: 10rem;
      font-size: xx-large;
      font-weight: bold;
      color: white;
    }

    .ball-container {
      display: flex;
    }
  </style>
</head>

<body>
  <h1>로또 추천 번호</h1>
  <button id="lotto-btn">행운 번호 받기</button>
  <div id="result"></div>
  <!-- 생성된 로또볼을 result div태그에 하나씩 add 할 것이다. -->

  <script src="https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js"></sc

  <script>
    const button = document.querySelector('#lotto-btn')
    button.addEventListener('click', function () {
```

```
    const ballContainer = document.createElement('div')
    ballContainer.classList.add('ball-container')

        // 랜덤한 숫자 6개 만들기
    const numbers = _.sampleSize(_.range(1, 46), 6) /*python range와 동일*/

    numbers.forEach((elemnet, idx) => { // 6개의 숫자중 한 숫자씩 뽑아서
      const ball = document.createElement('div') // div 태그 하나 만들고
      ball.classList.add('ball') // css 적용시켜 주고
      ball.innerText = numbers[idx] // 공 안의 값을 채워주고
      ball.style.backgroundColor = 'crimson' // 볼 색상 입혀서
      ballContainer.appendChild(ball) // 볼을 컨테이너에 넣어주기
    })

    // 생성한 로또번호 볼컨테이너를 result 태그의 맨 앞으로 붙이기
    const result = document.querySelector('#result')
    const firstchild = result.firstChild
    result.insertBefore(ballContainer, firstchild)
  })
 </script>
</body>

</html>
```



여기서 우리는 번호를 랜점하게 생성하기 위해

Javascript로 개발시 유용하게 사용하는 lodash라는 유틸리티를 사용하였다.

예> _.sampleSize(_.range(1, 46), 6)

lodash는 배열의 “깊은 복사” 또는 “객체 병합” 등과 같은 Javascript에 기본적으로 탑재되어 있지 않은 기능들을 제공해 주는 Javascript의 유용한 라이브러리 이다.

예시> 깊은복사

```
const obj = { a: 1, b: { c: 2 } };
const deepCopy = _.cloneDeep(obj);
deepCopy.b.c = 3; // 원본 obj.b.c는 여전히 2 가 됨
```

예시> 객체 병합

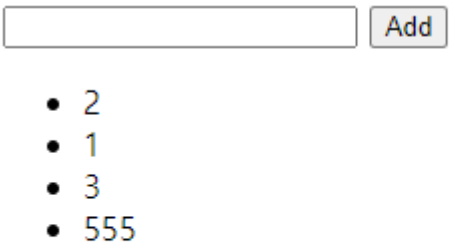
```
const test1 = { a: 1, b: { c: 2 } };
const test2 = { b: { d: 3 } };
const merged = _.merge(test1, test2); // { a: 1, b: { c: 2, d: 3 } }
```

그 다음은 TodoList 만들기를 살펴보자.

2. 클릭 눌러 todolist만들기

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Document</title>
</head>

<body>
  <form action="#">
```

```
    <input type="text" class="inputData">
    <input type="submit" value="Add">
  </form>
  <ul></ul>

  <script>
    const formTag = document.querySelector('form')// 먼저 form 태그를 타겟으로

    // 이벤트를 정의할 때에
    // formtag.addEventListener('submit',function(){}) 이렇게 했었는데
    // 콜백함수를 따로 분리해서 정의후 eventlistener를 나중에 사용하겠다.
    // 사실 개발시 이렇게 더 많이 쓴다.
    // 먼저 eventlistner에 들어갈 콜백 함수부터 만든다. (addTodo 라는 콜백함수를 만들

    const addTodo = function (event) {
      const inputTag = document.querySelector('.inputData')
      const data = inputTag.value //input창에 들어간 값을 가져오기

      if (data.trim()) {  // 공백 제거후 빈 문자열이 아니면 True
        const liTag = document.createElement('li')
        liTag.innerText = data // li태그 만들어 글자넣어주기

        const ulTag = document.querySelector('ul')
        ulTag.appendChild(liTag) // ul태그 자식으로 붙이기
      } else {
        alert('할일을 입력하세요')
      }
      event.target.reset() // form에 사용자가 입력한 값을 초기화 (지우기)
    }

    // 마지막으로 내가만든 콜백 함수를 가지고 이벤트를 먹인다. 부여한다.
    formTag.addEventListener('submit', addTodo)
  </script>
</body>

</html>
```



이상으로 Event 그리고 Event 취소에 대해서 살펴 보았다.

버블링

버블링이 무엇인지? 왜 중요 한지에 대해서 살펴보자.

버블링이란?

- 한 요소에 할당된 이벤트가 작동하면

  가장 최상단의 조상 요소를 만날 때까지 이벤트가 반복해서 진행(전파) 되는 것을 말한다.

버블링이 발생되는 코드를 작성해 보겠다.

예시>

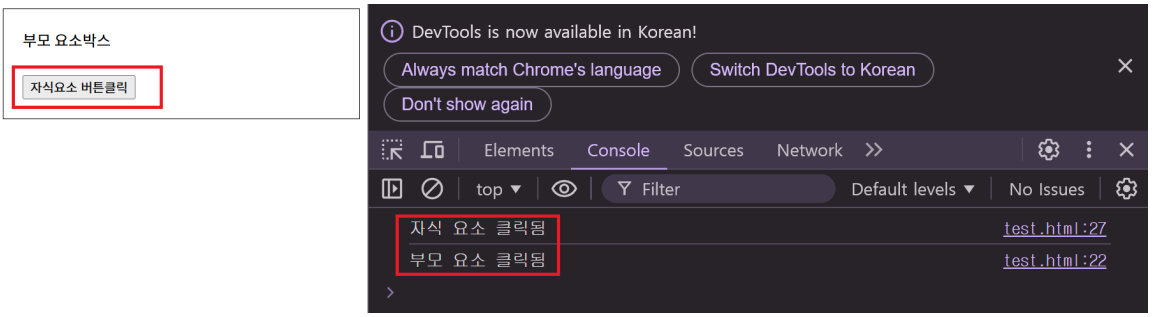
```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
```

```
  <div id="parent" style="padding: 20px; border: 1px solid black;">
    부모 요소박스 <br><br>
    <button id="child">자식요소 버튼클릭</button>
  </div>

  <script>
    const parent = document.getElementById('parent');
    const child = document.getElementById('child');

    // 부모 요소의 이벤트 리스너
    parent.addEventListener('click', function () {
      console.log('부모 요소 클릭됨');
    });

    // 자식 요소의 이벤트 리스너
    child.addEventListener('click', function (event) {
      console.log('자식 요소 클릭됨');
      // event.stopPropagation(); // 버블링을 중단하고 싶으면 이 줄을 사용할 수 있음
    });
  </script>
</body>

</html>
```

- div 태그가 부모 요소이고, 그 안에 button이 자식 요소 이다.
- 부모 요소에 클릭 이벤트 리스너를 추가하여 클릭 시 "부모 요소 클릭됨"이라고 출력되도록 하고
- 자식 요소에도 클릭 이벤트 리스너를 추가하여 클릭 시 "자식 요소 클릭됨"이라고 출력되도록 셋팅 했다.

- 버블링 확인을 위해서 자식요소 버튼 클릭이라는 button을 클릭 해보자.
- 콘솔창을 확인해 보면 버블링이 발생해서 자식 요소 클릭됨 뿐만 아니라

  부모 요소 클릭됨 까지 출력이 되었다.



- 자식요소의 이벤트가 DOM 트리 구조를 따라서 전파가 되었고, 그 결과 자식요소에서 발생한 이벤트가

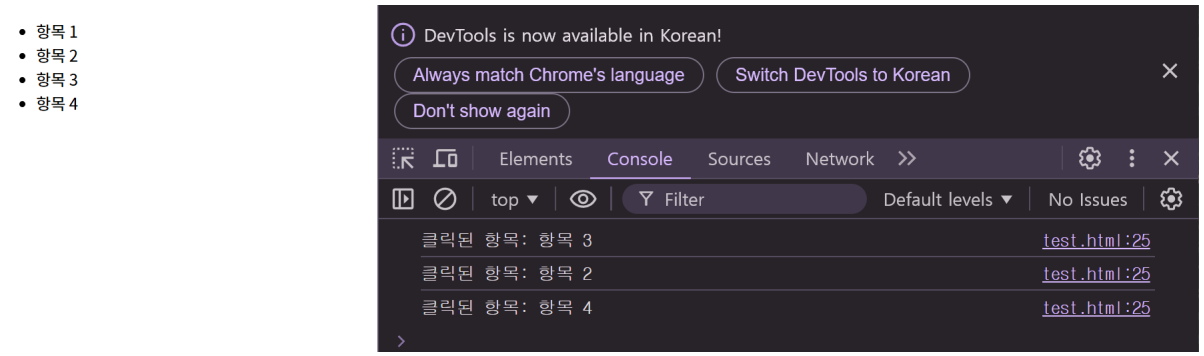
  부모요소로 올라가면서 이벤트가 처리 된 것이다.

- 이를 버블링이라고 하는데 버블링 덕분에 우리는 이벤트 리스너를 효율적으로 사용할 수 있다.

예시코드>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <ul id="itemList">
    <li>항목 1</li>
    <li>항목 2</li>
    <li>항목 3</li>
    <li>항목 4</li>
  </ul>

  <script>
    const itemList = document.getElementById('itemList');

    // 상위 요소인 ul에 이벤트 리스너를 등록
    itemList.addEventListener('click', function (event) {
```

```
    // 클릭된 요소가 li인지 확인
    if (event.target.tagName === 'LI') {
      console.log('클릭된 항목:', event.target.innerText);
    }
  });
</script>
</body>

</html>
```



- 리스트 항목이 클릭될 때마다 어떤 항목이 클릭되었는지를 알 수 있도록 해보았다.
- 먼저, ul 태그 안에 여러개의 li 태그들이 있다.
- ul 태그에 이벤트 리스너를 등록했다. 이때 모든 li 태그에 각각 리스너를 등록하는 대신 상위 태그인 ul 태그에 한번만 등록을 했다.
- 그 다음 클릭 이벤트가 발생하면, event.target을 사용해 실제 클릭된 요소를 확인했다.
- 클릭된 요소가 LI 인 경우 해당 항목의 텍스트를 콘솔에 출력했다.
    - JavaScript에서 `event.target.tagName`을 사용할 때는 항상 대문자로 반환 된다. 따라서 `event.target.tagName`의 값을 비교할 때는 항상 대문자로 작성해야 하기 때문에 “li”가 아닌 “LI”라고 작성한 점을 유의 깊게 보자.

이상으로 버블링 (이벤트 전파)를 살펴 보았다. 거의 대부분의 이벤트는 버블링 되나, focus 등 특정 몇몇 이벤트는 버블링이 발생하지 않는점도 기억해 두자.

마지막으로 `event.target`과 `event.currentTarget` 의 차이점에 대해서 살펴보고 마무리 하자.

아까 사용했던 예시코드 이다.

예시코드>

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>

<body>
  <ul id="itemList">
    <li>항목 1</li>
    <li>항목 2</li>
    <li>항목 3</li>
    <li>항목 4</li>
  </ul>

  <script>
    const itemList = document.getElementById('itemList');

    // 상위 요소인 ul에 이벤트 리스너를 등록
    itemList.addEventListener('click', function (event) {
      // 클릭된 요소가 li인지 확인
      if (event.target.tagName === 'LI') {
        console.log('event.target:', event.target); // 클릭된 요소
        console.log('event.currentTarget:', event.currentTarget); // 리스너가 등록된
      }
    });
  </script>
</body>

</html>
```

event.target

- 이벤트가 실제로 발생한 요소를 나타낸다.
- 따라서 여기서는 실제 클릭된 li 태그가 된다.

event.currentTarget

- 이벤트 리스너가 바인딩된 요소를 나타낸다.
- 따라서 여기서는 itemList인 ul 태그를 가르키게 된다.

이렇게 `event.target`과 `event.currentTarget`은 이벤트가 발생하는 상황에 따라 각각 다른 의미를 가지므로 이를 이해하는 것이 이벤트 처리에 매우 중요하다.

아까 위에서 input 이벤트를 설명하며 사용했던 예시로 한번 더 살펴보자.

```
<body>
  <input type="text" id="text-input">
  <p></p>
  <script>
    //1. input 선택
    const textInput = document.querySelector('#text-input')

    //2. input 이벤트 등록
    textInput.addEventListener('input', function (event) {

      console.log(event.target)        //input은 이벤트의 대상이다.

      console.log(event.target.value)   //input의 value 받아오기

      //3. input에 작성한 값을 p태그에 출력하기
      const pTag = document.querySelector('p')
      pTag.textContent = event.target.value
    })
  </script>
</body>
```

```
<body>
  <input type="text" id="text-input">
  <p></p>
  <script>
    //1. input 선택
    const textInput = document.querySelector('#text-input')

    //2. input 이벤트 등록
    textInput.addEventListener('input', function (event) {

      console.log(event.target)          //input은 이벤트의 대상이다.

      console.log(event.currentTarget.value)   //input의 value 받아오기

      //3. input에 작성한 값을 p태그에 출력하기
      const pTag = document.querySelector('p')
      pTag.textContent = event.currentTarget.value
    })
  </script>
</body>
```

이 경우에는

`event.target`과 `event.currentTarget`을 사용해도 결과는 동일하다. 그 이유는

`event.target`은 input 태그를 가르키고,

`event.currentTarget` 이것도 addeventListner로 input 이벤트를 등록 했기 때문에

둘다 input요소를 가르키게 된다. 그래서 아무거나 사용해도 무관한 상황이다.

< 끝 >

[참고]

addEventListener 에서 콜백함수는 특별하게 function 키워드의 경우 this가 의미 하는것은

addEventListener를 호출한 대상을 (event.target)을 뜻한다.

addEventListener 에서 콜백함수는로써 화살표 함수를 사용하면 this 가 의미하는 것은

window 를 의미한다. 화살표 함수는 this라는 것이 없다. 그래서 외부함수에서의 this 할만한 것을 가져오는데 그때 가져오는 것이 window 이다.

this 때문에 머리가 조금 아플 수도 있다. 그 이유는 화살표 함수는 this라는 개념이 명확하게 없기

때문에 상황에 따라 this의 범위가 다 달라진다. 그래서 모든 상황마다 this가 가르키는 것을 찾기

위해 공부하면서 스스로 특별한 규칙 또는 로직을 정의하는 것은 큰 의미가 없다.

그렇지만, 우리가 잘 기억해야 할 것은 분명하다.

화살표 함수를 사용을 언제하고, 언제 하지 않는 것이 좋은지 만을 잘 구분하고 있으면 된다.

결론이다.

[중요] 화살표 함수 사용을 피해야 할 때 그리고 주의사항

1. 화살표 함수 사용하기 위해 함수를 호출하기 전에 정의를 먼저 해줘야 한다. 화살표 함수는 호이스팅이 발생하지 않기 때문이다.

2. 객체 안에 메서드의 용도로 함수 사용 시 화살표 함수 사용하지 말자.

   fuction 사용하면 이때의 this는 해당 객체를 의미 하지만

   화살표 함수 사용 시 this는 window를 가르키기 때문이다.

3. addEventListener 사용시 콜백함수로 화살표 함수를 사용하지 말자.

   fuction 사용하면 이때의 this는 이벤트 타겟을 의미 하지만

   화살표 함수 사용 시 this는 window를 가르키기 때문이다.

4. 생성자함수 생성시 화살표 함수를 사용하지 말자.

   function 사용하자. 화살표함수는 인스턴스를 생성 할 수가 없기 때문이다.

[중요] 화살표 함수 사용이 권장 될 때

1. 간단하게 일반함수 호출을 작성하고 싶을 때 화살표 함수로 간결하게 작성이 가능하다.

2. 중첩된 함수에서 this를 사용할 때

   a. 헬퍼메서드(forEach, map)에서 인자로 콜백함수를 화살표 함수로 작성한다. 이때 this는 해당 객체를 의미하기 때문이다.

b. 다음 시간에 배울 setTimeout 함수에 인자로 콜백함수를 화살표 함수로 작성한다. 이때 this도 해당 객체를 의미한다.