이벤트

Introduction

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('cli
      alert('Clicked!');
    });
  </script>
</body>
</html>
```

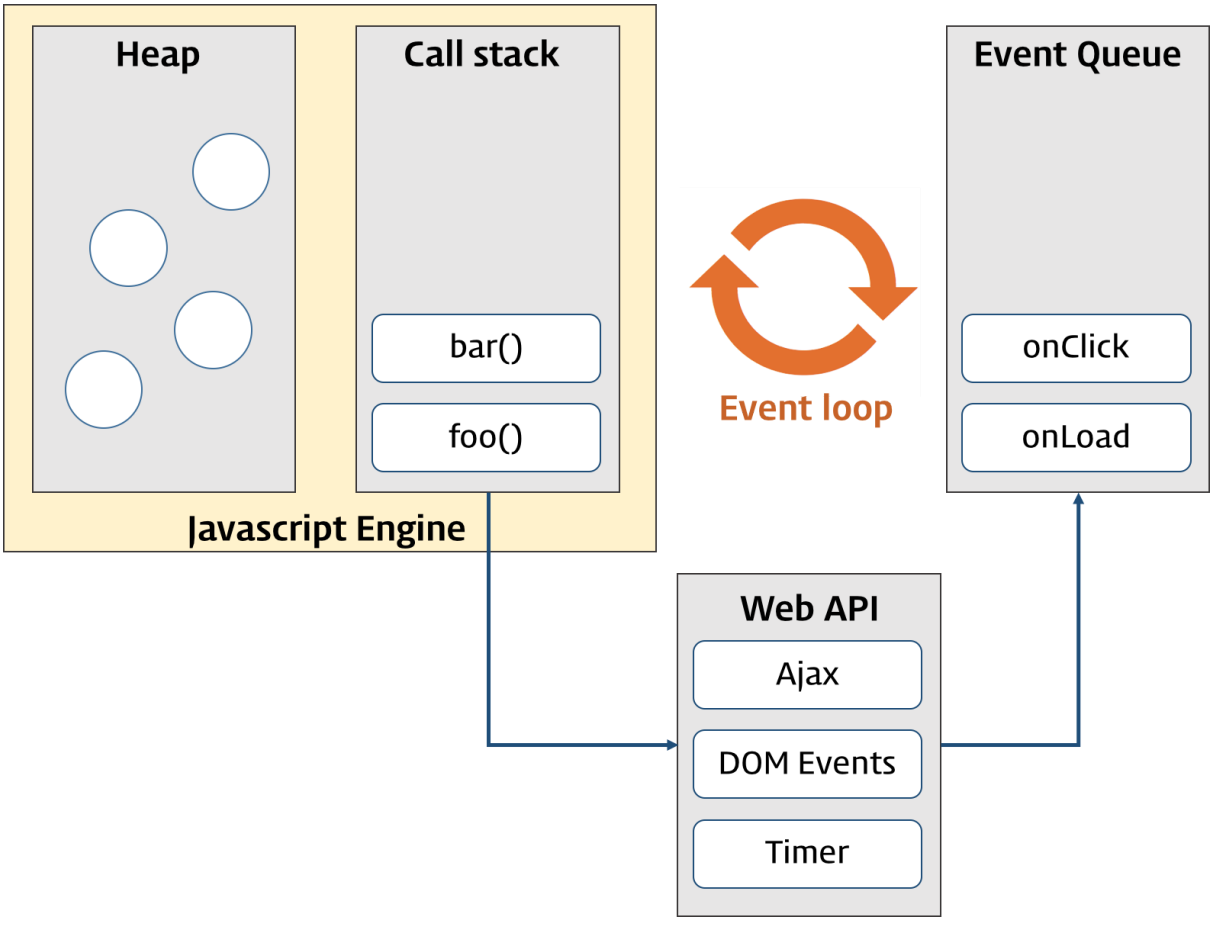
이벤트가 발생하면 그에 맞는 반응을 하여야 한다. 이를 위해 이벤트는 일반적으로 함수에 연결되며 그 함수는 이벤트가 발생하기 전에는 실행되지 않다가 이벤트가 발생되면 실행된다. 이 함수를 **이벤트 핸들러**라 하며 이벤트에 대응하는 처리를 기술한다.

이벤트 루프(Event Loop)와 동시성(Concurrency)

브라우저는 단일 쓰레드(single-thread)에서 이벤트 드리븐(event-driven) 방식으로 동작한다.

단일 쓰레드는 쓰레드가 하나뿐이라는 의미이며 이말은 곧 하나의 작업(task)만을 처리할 수 있다는 것을 의미한다. 하지만 실제로 동작하는 웹 애플리케이션은 많은 task가 동시에 처리되는 것처럼 느껴진다. 이처럼 자바스크립트의 동시성(Concurrency)을 지원하는 것이 바로 **이벤트 루프(Event Loop)**이다.

브라우저의 환경을 그림으로 표현하면 아래와 같다.



구글의 V8을 비롯한 대부분의 자바스크립트 엔진은 크게 2개의 영역으로 나뉜다.

✍🏻 **Call Stack(호출 스택)**

작업이 요청되면(함수가 호출되면) 요청된 작업은 순차적으로 Call Stack에 쌓이게 되고 순차적으로 실행된다. 자바스크립트는 단 하나의 Call Stack을 사용하기 때문에 해당 task가 종료하기 전까지는 다른 어떤 task도 수행될 수 없다.

✍🏻 **Heap**

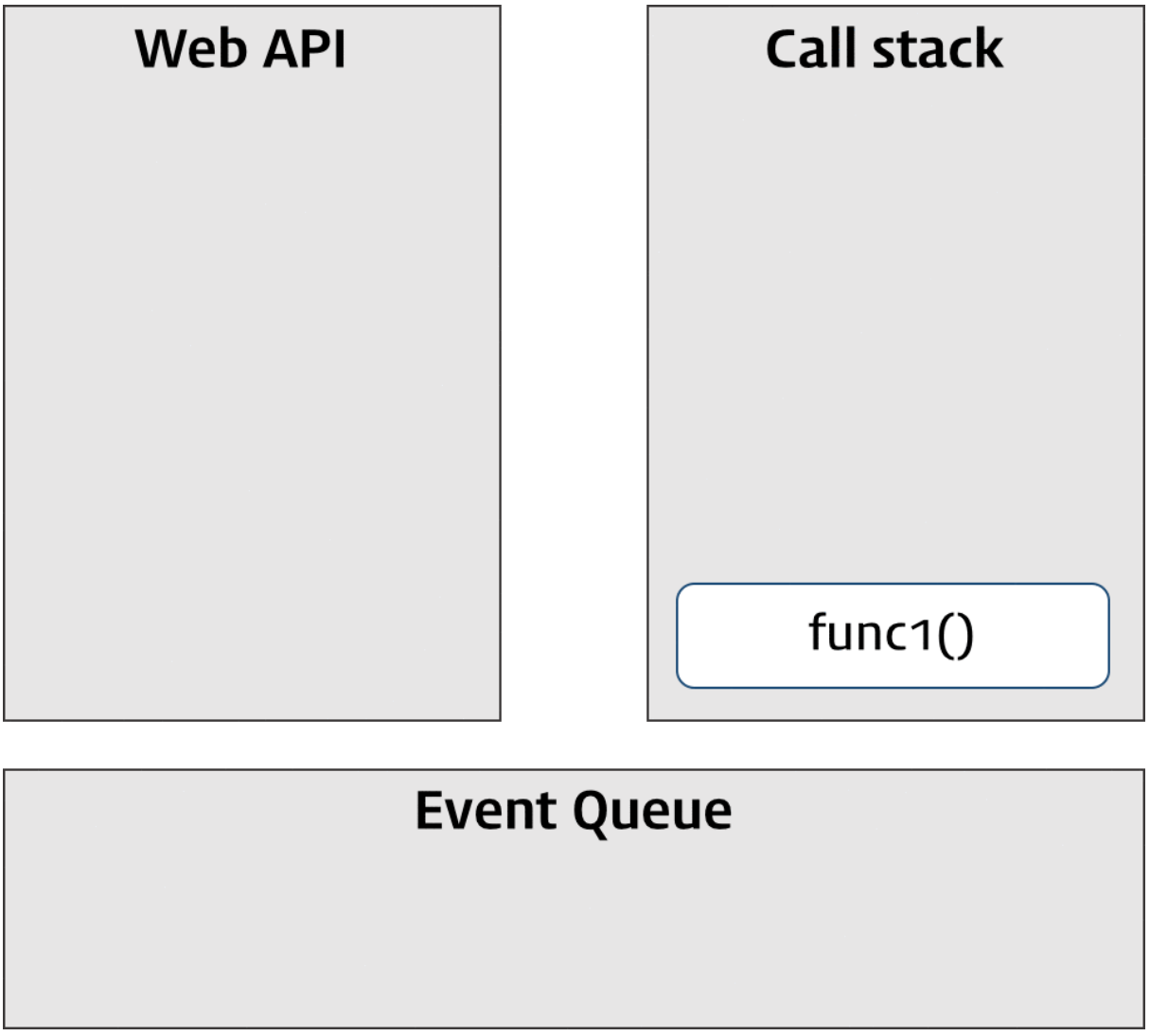
동적으로 생성된 객체 인스턴스가 할당되는 영역이다.

아래의 예제가 어떻게 동작할지 살펴보자.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  setTimeout(function () {
    console.log('func2');
  }, 0);

  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

함수 func1이 호출되면 함수 func1은 Call Stack에 쌓인다. 그리고 함수 func1은 함수 func2을 호출하므로 함수 func2가 Call Stack에 쌓이고 setTimeout가 호출된다. **setTimeout의 콜백함수는 즉시 실행되지 않고 지정 대기 시간만큼 기다리다가 "tick"**

이벤트가 발생하면 태스크 큐로 이동한 후 Call Stack이 비어졌을 때 Call Stack으로 이동되어 실행된다.



DOM 이벤트 핸들러도 이와 같이 동작한다.

```
function func1() {
  console.log('func1');
  func2();
}

function func2() {
  // <button class="foo">foo</button>
```

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

  elem.addEventListener('click', function () {
    this.style.backgroundColor = 'indigo';
    console.log('func2');
  });

  func3();
}

function func3() {
  console.log('func3');
}

func1();
```

함수 func1이 호출되면 함수 func1은 Call Stack에 쌓인다. 그리고 함수 func1은 함수 func2을 호출하므로 함수 func2가 Call Stack에 쌓이고 addEventListener가 호출된다. **addEventListener의 콜백함수는 foo 버튼이 클릭되어 click 이벤트가 발생하면 태스크 큐로 이동한 후 Call Stack이 비어졌을 때 Call Stack으로 이동되어 실행된다.**

이벤트의 종류

UI Event

Event	Description
load	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
keyup	누르고 있던 키를 뗄 때
keypress	키를 누르고 뗏을 때

Mouse Event

Event	Description
click	마우스 버튼을 클릭했을 때
dbclick	마우스 버튼을 더블 클릭했을 때
mousedown	마우스 버튼을 누르고 있을 때
mouseup	누르고 있던 마우스 버튼을 뗄 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
focus/focusin	요소가 포커스를 얻었을 때
blur/foucusout	요소가 포커스를 잃었을 때

Form Event

Event	Description
input	input 또는 textarea 요소의 값이 변경되었을 때

Event	Description
	contenteditable 어트리뷰트를 가진 요소의 값이 변경되었을 때
change	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 핸들러 등록

인라인 이벤트 핸들러 방식

HTML 요소의 이벤트 핸들러 어트리뷰트에 이벤트 핸들러를 등록하는 방법이다.

```
<!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 분

리하는 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</but
  <script>
    function myHandler1() {
      alert('myHandler1');
    }
    function myHandler2() {
      alert('myHandler2');
    }
  </script>
```

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

이벤트 핸들러 프로퍼티 방식

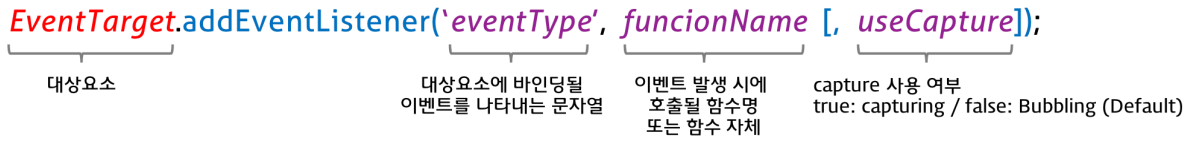
인라인 이벤트 핸들러 방식처럼 HTML과 Javascript가 뒤섞이는 문제는 해결할 수 있는 방식이다. 하지만 이벤트 핸들러 프로퍼티에 하나의 이벤트 핸들러만을 바인딩할 수 있다는 단점이 있다.

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

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

    // 두번째 바인딩된 이벤트 핸들러
    btn.onclick = function () {
      alert('① Button clicked 2');
    };

    // addEventListener 메소드 방식
    // 첫번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('② Button clicked 1');
    });

    // 두번째 바인딩된 이벤트 핸들러
    btn.addEventListener('click', function () {
      alert('② Button clicked 2');
```

```
    });
  </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.



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 이상이야한다는 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 ㄷ
elem.addEventListener('click', foo);      // 이벤트 발생 시까지 ㄷ
```

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 **this는 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 **this는 이벤트에 바인딩된 요소**를 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">Butto
      console.log(this === e.currentTarget); // true
    };
  </script>
```

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

addEventListener 메소드 방식

addEventListener 메소드에서 지정한 이벤트 핸들러는 콜백 함수이지만 이벤트 핸들러 내부의 **this는 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">Butto
      console.log(this === e.currentTarget); // true
    });
  </script>
</body>
</html>
```

이벤트의 흐름

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 캡처**

링부터 시작하여 버블링으로 종료한다는 것이다. 즉, 이벤트가 발생했을 때 캡처링과 버블링은 순차적으로 발생한다.

캡처링은 IE8 이하에서 지원되지 않는다.

addEventListener 메소드의 세번째 매개변수에 true를 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: ce
    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
    // 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', ha
  document.querySelector('body').addEventListener('click', ha

  document.querySelector('div.top').addEventListener('click',
  document.querySelector('div.middle').addEventListener('clic
  document.querySelector('div.bottom').addEventListener('clic
  </script>
</body>
</html>
```

좀 더 자세히 살펴보자. 먼저 버블링의 경우 어떻게 동작하는지 알아본다.

```
<!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
    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, 캡쳐링 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.
```

캡처링의 경우 어떻게 동작하는지 살펴보자.

```
<!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.
```

다음은 캡처링과 버블링이 혼용되는 경우이다.

```
<!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>
```

와 같이 event 객체는 이벤트 핸들러에 암묵적으로 전달된다. 그러나 이벤트 핸들러를 선언할 때, event 객체를 전달받을 첫번째 매개변수를 명시적으로 선언하여야 한다. 예제에서 e 라는 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('clic
k', hide);
    document.getElementById('btn2').addEventListener('clic
k', 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. 따
      // 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 this는 이벤트에 바인딩된 DOM 요소를 가리키며 이것은 이벤트 객체의 currentTarget 프로퍼티와 같다. 따라서 이벤트 핸들러 함수 내에서 currentTarget과 this는 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 요
      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 상
      this.style.backgroundColor = '#A5D9F3';
    }

    // div 요소에 이벤트 핸들러가 바인딩되어 있다.
    // 자식 요소인 button이 발생시킨 이벤트가 버블링되어 div 요소에도 전
    // 따라서 div 요소에 이벤트 핸들러가 바인딩되어 있으면 자식 요소인 bu
    document.querySelector('div').addEventListener('click', b
  </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}
  }

  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	캡쳐링 단계
2	타깃
3	버블링 단계

Event Delegation (이벤트 위임)

우선 아래 예제를 살펴보자.

```
<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>
```

모든 li 요소가 클릭 이벤트에 반응하는 처리를 구현하고 싶은 경우, li 요소에 이벤트 핸들러를 바인딩하면 총 6개의 이벤트 핸들러를 바인딩하여야 한다.

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

document.querySelector('#post-1').addEventListener('click', p
```

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

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>
```

기본 동작의 변경

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', funct
    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
  <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
  <script>

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

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

  // pure js
  document.querySelector('a').addEventListener('click', funct
    // 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.