이벤트 기초 1

브라우저 이벤트 소개

- 이벤트(event) 는 무언가 일어났다는 신호이며 모든 DOM 노드는 이런 신호를 만들어 낼 수 있음

자주 사용되는 유용한 DOM 이벤트의 종류

```
# 마우스 이벤트

click - 요소 위에서 마우스 왼쪽 버튼을 눌렀을 때(터치스크린이 있는 장치에선 탭 했을 때) 발생
contextmenu - 요소 위에서 마우스 오른쪽 버튼을 눌렀을 때 발생
mouseover와 mouseout - 마우스 커서를 요소 위로 움직였을 때, 커서가 요소 밖으로 움직였을 때 발생
mousedown과 mouseup - 요소 위에서 마우스 왼쪽 버튼을 누르고 있을 때, 마우스 버튼을 뗄 때 발생
mousemove - 마우스를 움직일 때 발생

# 폼 요소 이벤트

submit - 사용자가 폼을 제출할 때 발생
focus - 사용자가 input과 같은 요소에 포커스 할 때 발생

# 키보드 이벤트

keydown과 keyup - 사용자가 키보드 버튼을 누르거나 뗄 때 발생

# 문서 이벤트

DOMContentLoaded - HTML이 전부 로드 및 처리되어 DOM 생성이 완료되었을 때 발생

# CSS 이벤트

transitionend - CSS 애니메이션이 종료되었을 때 발생
```

이벤트 핸들러

- 이벤트에 반응하려면 **이벤트가 발생했을 때 실행되는 "함수"인 핸들러(handler)** 를 할당해야 함
    - 핸들러는 사용자의 행동에 따라 발생한 이벤트에 **어떻게 반응할지를 자바스크립트 코드(=함수)로 표현**한 것
        - 즉, 이벤트를 처리하기 위한 함수 => 이벤트 핸들러 or 이벤트 핸들러 함수 or 그냥 핸들러

여러 방법을 이용한 이벤트 핸들러 할당

1. HTML 안의 on<event> 속성에 핸들러를 할당

```
<input value="클릭해 주세요." onclick="alert('클릭!')" type="button">
```

- 버튼을 클릭하면 onclick 안의 코드가 실행
  - 긴 코드를 HTML 속성값으로 사용하는 것은 권장되지 않음
    - **이유1)** HTML 태그 중간에 자바스크립트 코드가 들어가 있으면 관리하기 불편함
    - **이유2)** 긴 코드를 작성하기가 어려움
    - 따라서 정말 간단한 한 줄 짜리 코드를 사용하는 것이 아니라면, **함수를 만들어서 이를 호출**하는 방법이 권장됨

함수(이벤트 핸들러)를 이용하기

```
<script>
function countRabbits() {
    for(let i=1; i<=3; i++) {
        alert(`토끼 ${i}마리`);
    }
}
</script>

<input type="button" onclick="countRabbits()" value="토끼를 세봅시다!">
```

- HTML 속성은 대소문자를 구분하지 않기 때문에, ONCLICK은 onClick이나 onCLICK과 동일하게 작동하지만, 속성값은 **보통 onclick 같이 소문자로 작성**함

2. DOM 프로퍼티에 핸들러 대입

```
<input id="elem" type="button" value="클릭해 주세요.">
<script>
elem.onclick = function() { alert('감사합니다.'); };
</script>
```

- onclick 프로퍼티는 단 하나밖에 없기 때문에 **여러개의 이벤트 핸들러를 할당할 수 없음**을 유의
- 핸들러를 제거하고 싶다면 `elem.onclick = null` 같이 null 값을 할당
- 이벤트 핸들러 내부에서 **this 값을 통해 이벤트가 발생한 요소에 접근** 가능

```
<!-- 아래 예시의this.innerHTML에서 this는 button이므로 버튼을 클릭하면 버튼 안의 콘텐츠가 얼럿창에 출력 -->
<button onclick="alert(this.innerHTML)">클릭해 주세요.</button>
```

- setAttribute 메서드를 통한 핸들러 할당은 불가능함

```
// <body>를 클릭하면 에러가 발생 (속성은 항상 문자열이기 때문에, 함수가 문자열이 되어버리기 때문)
document.body.setAttribute('onclick', function() { alert(1) });
```

addEventListener, removeEventListener (***)

- HTML 속성과 DOM 프로퍼티를 이용한 이벤트 핸들러 할당 방식을 사용할 경우, **하나의 이벤트에 복수의 핸들러를 할당할 수 없다는 문제가 발생**함
- 따라서, **addEventListener 와 removeEventListener 라는 특별한 메서드를 이용해 핸들러를 추가, 제거**할 수 있는 방법이 제공되었으며 이를 통해서 핸들러를 여러 개 할당할 수 있음
    - 하나의 핸들러만 할당해도 좋다면 이전 방법을 사용해도 무관함

```
element.addEventListener(event, handler, [options]);
```

addEventListener 전달 인자의 용도

```
event : 이벤트 이름(예: "click", HTML 속성 이름에서 "on"을 제거한 문자열을 전달)
handler : 이벤트 핸들러 함수
```

options 값으로 전달할 객체의 내용

```
once : true이면 이벤트가 트리거 되는 시점에 리스너가 자동으로 삭제됨 (즉, 일회용 리스너를 추가하기 위해서 사용)
capture : 어느 단계에서 이벤트를 다뤄야 하는지를 알려주는 프로퍼티, 호환성 유지를 위해 options를 객체가 아닌 false/true로 할당하는 것도 가능한데, 이는 { capture: false/true }는 와 동일
passive : true이면 리스너에서 지정한 함수가 preventDefault 메소드를 호출하지 않음
```

- removeEventListener 메서드를 통한 리스너 제거는 **동일한 참조를 가진 함수를 대상으로 진행**해야 함을 유의
    - 즉, 핸들러를 삭제하려면 **핸들러 할당 시 사용한 함수를 그대로 전달**해주어야 함

```
elem.addEventListener( "click" , () => alert('감사합니다!'));
// (함수 내용은 같지만) 참조가 다른, 서로 다른 함수이므로 이벤트 핸들러 함수가 제거되지 않음
elem.removeEventListener( "click", () => alert('감사합니다!'));
```

```
function handler() {
    alert('감사합니다!');
}

input.addEventListener("click", handler);
// 참조가 같으므로 이벤트 핸들러 함수가 제거됨
input.removeEventListener("click", handler);
```

이벤트 객체

- 발생한 이벤트를 처리하기 위해서는 어떤 일이 일어났는지 상세히 알아야 함
    - 가령, 클릭 이벤트가 발생했다면 **마우스 포인터의 위치**가 어디인지, 키 입력 이벤트가 발생했다면 **어떤 키가 입력**되었는지 등에 대한 상세한 정보가 필요할 수 있음
    - **이벤트 타입에 따라 이벤트 객체에서 제공하는 프로퍼티는 다를 수 있음**을 유의 (마우스 이벤트에서 필요한 정보와 키보드 이벤트에서 필요한 정보가 다를 수 있으므로)
        - 단, 이벤트 종류를 나타내는 type, 이벤트가 발생한 요소를 가리킬 currentTarget과 같은 **공통 프로퍼티**도 존재함

- 이벤트가 발생하면 브라우저는 **이벤트 객체(event object)**라는 것을 만들고 객체에 이벤트에 관한 상세한 정보를 저장한 후, 핸들러에 인수 형태로 전달함

```
<input type="button" value="클릭해 주세요." id="elem">

<script>
// 함수의 전달 인자 목록에 event 인자를 추가
elem.onclick = function(event) {
    // 이벤트 타입과 요소, 클릭 이벤트가 발생한 좌표를 보여주기 (이벤트 객체 공통 속성)
    alert(event.type + " 이벤트가 " + event.currentTarget + "에서 발생했습니다.");

    // 특정 이벤트(마우스 클릭)와 관련된 속성 접근
    alert("이벤트가 발생한 곳의 좌표는 " + event.clientX + ":" + event.clientY +"입
니다.");
};
</script>
```

객체 형태의 핸들러와 handleEvent

- addEventListener를 사용하면 함수뿐만 아니라 객체를 이벤트 핸들러로 할당할 수 있음
    - 이 경우 이벤트가 발생하면 객체에 구현한 **handleEvent 메서드가 호출**됨

```
<button id="elem">클릭해 주세요.</button>

<script>
let obj = {
    handleEvent(event) {
        alert(event.type + " 이벤트가 " + event.currentTarget + "에서 발생했습니
다.");
    }
};

elem.addEventListener('click', obj);
</script>
```

- 클래스를 사용할 수도 있음 (handleEvent 메서드 정의) (TODO)

버블링과 캡처링

버블링

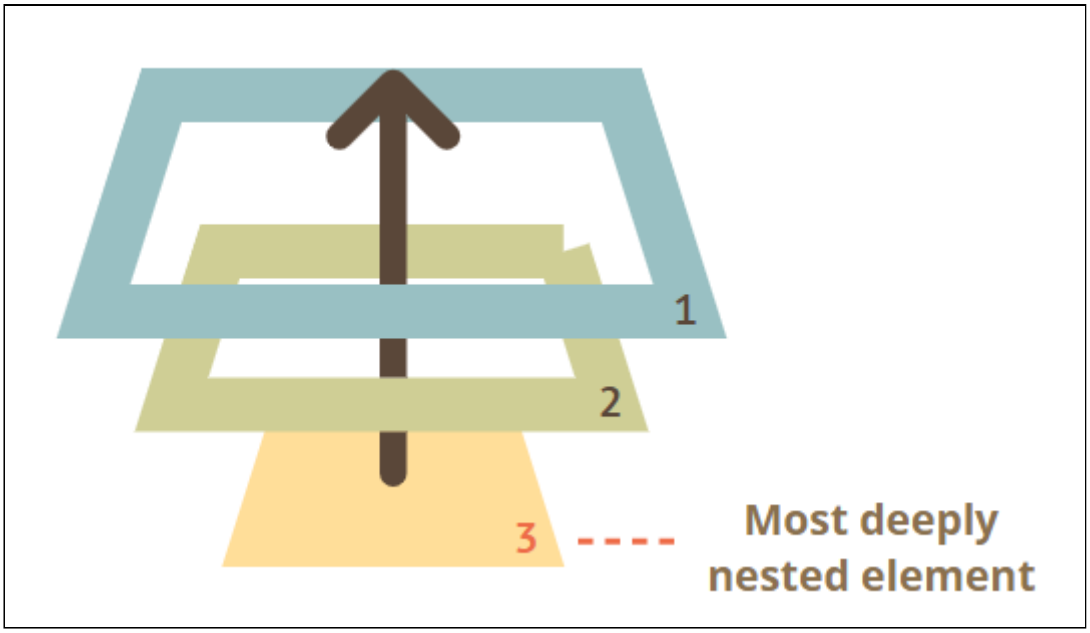
- 한 요소에 이벤트가 발생하면, 이 요소에 할당된 핸들러가 동작하고, 이어서 부모 요소의 핸들러가 동작합니다. 가장 최상단의 조상 요소를 만날 때까지 이 과정이 반복되면서 요소 각각에 할당된 핸들러가 동작되는 방식

```
<style>
body * {
    margin: 10px;
    border: 1px solid blue;
}
</style>

<form onclick="alert('form')">FORM
    <div onclick="alert('div')">DIV
        <p onclick="alert('p')">P</p>
    </div>
</form>
```

- 가장 안쪽의 p 요소를 클릭하면 순서대로 다음과 같은 일이 벌어집니다.

```
1. <p>에 할당된 onclick 핸들러가 동작
2. 바깥의 <div>에 할당된 핸들러가 동작
3. 그 바깥의 <form>에 할당된 핸들러가 동작
4. document 객체를 만날 때까지, 각 요소에 할당된 onclick 핸들러가 동작
```



- 이런 흐름을 '이벤트 버블링'이라고 부릅니다. 이벤트가 제일 깊은 곳에 있는 요소에서 시작해 부모 요소를 거슬러 올라가며 발생하는 모양이 마치 물속 거품(bubble)과 닮았기 때문
- 이벤트가 발생한 가장 안쪽의 요소는 타깃(target) 요소라고 불리고, event.target을 사용해 접근할 수 있습
- event.target과 this(=event.currentTarget)는 다음과 같은 차이점이 존재

```
event.target은 실제 이벤트가 시작된 '타깃' 요소입니다. 버블링이 진행되어도 변하지 않음
this는 '현재' 요소로, 현재 실행 중인 핸들러가 할당된 요소를 참조
```

버블링 중단하기

- 이벤트 버블링은 타깃 요소에서 시작해서 html 요소를 거쳐 document 객체를 만날 때까지 각 노드에서 모두 발생함

- 이 과정에서 핸들러에게 이벤트를 완전히 처리하고 난 후 **버블링을 중단하도록 명령**할 수 있음
    - 이벤트 객체의 **stopPropagation 메서드**를 호출하면 상위 요소로의 이벤트 버블링이 일어나지 않게 됨

```
<body onclick="alert(`버블링은 여기까지 도달하지 못합니다.`)">
    <button onclick="event.stopPropagation()">클릭해 주세요.</button>
</body>
```

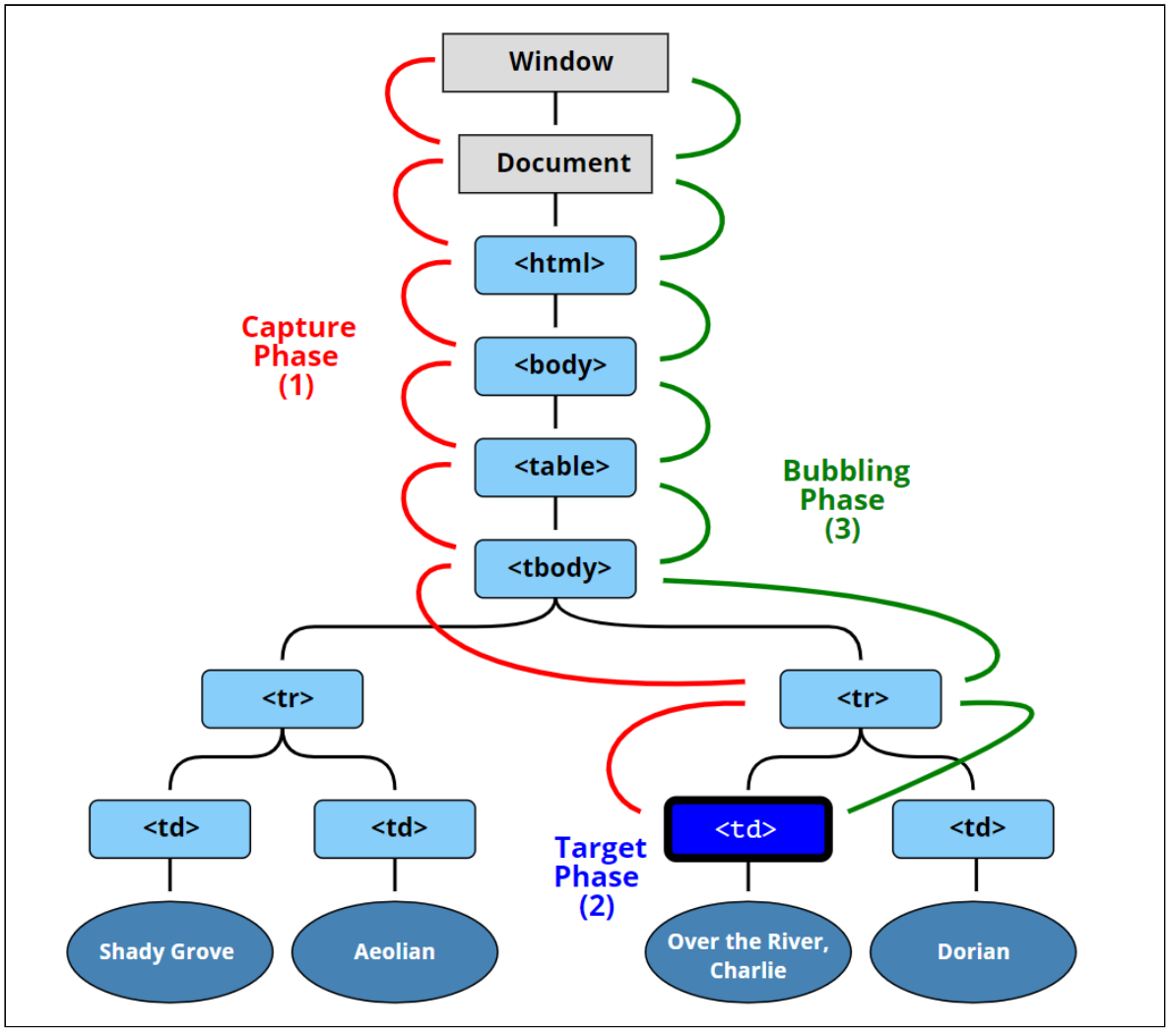
https://stackoverflow.com/questions/5299740/stoppropagation-vs-stopimmediatepropagation

캡처링 (@)

- 표준 DOM 이벤트에서 정의한 이벤트 흐름에는 **3가지의 단계가 존재**함

```
캡처링 단계: 이벤트가 "하위 요소"로 전파되는 단계
타깃 단계: 이벤트가 "실제 타깃 요소"에 전달되는 단계
버블링 단계: 이벤트가 "상위 요소"로 전파되는 단계
```

- 가령, td 요소를 클릭했다고 가정하면 밑의 그림과 같이 이벤트가
    - **최상위 조상에서 시작해 아래로 전파**되고 (캡처링 단계)
    - 이벤트가 **타깃 요소에 도착해 실행**된 후 (타깃 단계)
    - **다시 위로 전파**됨 (버블링 단계)



- 캡처링 단계에서 이벤트를 잡아내려면 addEventListener의 **capture 옵션을 true로 설정**

```
elem.addEventListener(..., {capture: true})
// 아니면, 아래 같이 {capture: true} 대신, true를 써줘도 됩니다.
elem.addEventListener(..., true)
```