JavaScript 101 - (8) 이벤트의 버블링과 캡처링

이벤트의 개념과 종류

- 이벤트란 웹 페이지에서 발생하는 사건을 의미한다.
- 예를 들어 마우스 클릭, 키보드 입력, 스크롤 등이 있다.
- 이벤트에 따라 다른 동작이 실행될 수 있다.

이벤트 핸들러와 이벤트 모델

- 이벤트 핸들러란 이벤트가 발생했을 때 실행되는 함수나 코드 조각이다.
- 이벤트 핸들러는 HTML 요소에 직접 지정하거나(인라인 방식), 자바스크립트에서 할당하거나(전통적 방식), `addEventListener` 메서드를 사용하여 등록할 수 있다(표준 방식).
- 이벤트 모델이란 여러 요소에 동일한 이벤트가 발생했을 때 어떻게 처리되는지 정하는 규칙이다.
- 표준 방식에서는 `addEventListener` 메서드의 세 번째 인자로 이벤트 모델을 선택할 수 있다.

버블링과 캡처링

인터넷 브라우저에서는 이벤트가 전파되는 두 가지 방식, 버블링(bubbling)과 캡처링(capturing)이 있다. 버블링은 자식 요소에서 부모 요소로 전파되는 방식이고, 캡처링은 부모 요소에서 자식요소로 전파되는 방식이다.

- 버블링은 이벤트가 발생한 요소부터 상위 요소로 순차적으로 전파되는 방식입니다. 예를 들어 `div` 요소를 클릭하면 `div`의 클릭 이벤트 핸들러가 실행되고, 그 다음 부모 요소인 `body`의 클릭 이벤트 핸들러가 실행됩니다.
- 캡처링은 버블링과 반대로 최상위 요소부터 하위 요소로 순차적으로 전파되는 방식입니다. 예를 들어 `div` 요소를 클릭하면 먼저 `body`의 클릭 이벤트 핸들러가 실행되고, 그 다음 자식 요소인 div의 클릭 이벤트 핸들러가 실행됩니다.
- 버블링과 캡처링을 선택하는 방법은 `addEventListener` 메서드의 세 번째 인자로 `true` 또는 `false`를 전달하는 것입니다. `true`면 캡처링을 사용하고, `false`면 버블링을 사용합니다. 기본값은 `false`입니다.
- 버블링과 캡처링을 막는 방법은 `event.stopPropagation` 메서드를 호출하는 것입니다. 이 메서드는 현재 이벤트가 상위 또는 하위 요소로 전파되지 않도록 합니다.

버블링(bubbling)

JavaScript에서 버블링이란 한 요소에 이벤트가 발생하면 해당 요소의 이벤트 핸들러가 동작하고, 그 다음 부모 요소의 이벤트 핸들러가 동작하는 것을 말합니다. 예를 들어, div 태그 안에 button 태그가 있고 둘 다 클릭 이벤트를 갖고 있다면, button을 클릭하면 button의 클릭 이벤트와 div의 클릭 이벤트가 모두 실행됩니다.

버블링의 예제코드는 다음과 같습니다.

```
// HTML
<div class="div">
  <button class="button">Click me</button>
</div>

// JavaScript
const div = document.querySelector(".div");
const button = document.querySelector(".button");

div.addEventListener("click", () => {
  console.log("div clicked");
});

button.addEventListener("click", () => {
  console.log("button clicked");
});
```

이 코드를 실행하면 button을 클릭하면 콘솔에 "button clicked"와 "div clicked"가 차례로 출력됩니다. 이것이 버블링의 현상입니다.

버블링을 방지하고 싶다면, event 객체의 `stopPropagation` 메서드를 사용할 수 있습니다. 예를 들어,

```
// HTML
<div class="div">
  <button class="button">Click me</button>
</div>

// JavaScript
const div = document.querySelector(".div");
const button = document.querySelector(".button");

div.addEventListener("click", () => {
  console.log("div clicked");
});

button.addEventListener("click", (event) => {
  console.log("button clicked");
  event.stopPropagation(); // 버블링 방지
});
```

이 코드를 실행하면 button을 클릭하면 콘솔에 "button clicked"만 출력되고 "div clicked"는 출력되지 않습니다. stopPropagation 메서드는 현재 요소에서만 이벤트 핸들러를 실행하고 부모 요소로 전파되는 것을 막습니다.

캡처링(capturing)

캡처링이란 이벤트가 발생하면 최상위 요소부터 시작해서 해당 요소까지 이벤트 핸들러가 차례로 실행되는 것을 말합니다. 예를 들어, div 태그 안에 button 태그가 있고 둘 다 클릭 이벤트를 갖고 있다면, button을 클릭하면 div의 클릭 이벤트와 button의 클릭 이벤트가 모두 실행됩니다.

캡처링의 예제코드는 다음과 같습니다.

```
// HTML
<div class="div">
  <button class="button">Click me</button>
</div>

// JavaScript
const div = document.querySelector(".div");
const button = document.querySelector(".button");

div.addEventListener("click", () => {
  console.log("div clicked");
}, true); // 캡처링 옵션 활성화

button.addEventListener("click", () => {
  console.log("button clicked");
}, true); // 캡처링 옵션 활성화
```

이 코드를 실행하면 button을 클릭하면 콘솔에 "div clicked"와 "button clicked"가 차례로 출력됩니다. 이것이 캡처링의 현상입니다.

캡처링을 방지하고 싶다면, addEventListener 메서드의 세 번째 인자로 false를 전달할 수 있습니다. 예를 들어,

```
// HTML
<div class="div">
  <button class="button">Click me</button>
</div>

// JavaScript
const div = document.querySelector(".div");
const button = document.querySelector(".button");

div.addEventListener("click", () => {
  console.log("div clicked");
}, false); // 캡처링 옵션 비활성화

button.addEventListener("click", () => {
  console.log("button clicked");
}, false); // 캡처링 옵션 비활성화
```

이 코드를 실행하면 button을 클릭하면 콘솔에 "button clicked"와 "div clicked"가 차례로 출력됩니다. 이것은 버블링의 현상입니다.

캡처링과 버블링의 유용한 상황

캡처링과 버블링은 이벤트가 발생한 노드를 찾기 위해 DOM 트리를 탐색하는 방식입니다. 캡처링은 도큐먼트 루트에서부터 타겟 노드까지 내려가는 방식이고, 버블링은 타겟 노드에서부터 도큐먼트 루트까지 올라오는 방식입니다.

캡처링과 버블링은 다음과 같은 상황에서 유용할 수 있습니다.

- 캡처링은 이벤트가 발생하기 전에 어떤 작업을 수행하고 싶을 때 사용할 수 있습니다. 예를 들어, 부모 요소에 캡처링 단계에서 실행되는 이벤트 리스너를 등록하면 자식 요소에 발생한 이벤트를 가로채거나 수정할 수 있습니다.
- 버블링은 이벤트 위임 패턴을 구현할 때 사용할 수 있습니다. 예를 들어, 여러 자식 요소에 공통된 이벤트 리스너를 부모 요소에 한 번만 등록하면 자식 요소에 발생한 이벤트가 부모 요소로 전달되면서 실행됩니다. 이렇게 하면 메모리와 성능을 절약할 수 있습니다.

이벤트 위임

이벤트 위임이란 하위 요소마다 이벤트 리스너를 등록하지 않고 상위 요소에서 하위 요소의 이벤트를 제어하는 방식입니다. 이벤트 위임을 사용하면 다음과 같은 장점이 있습니다.

- 메모리와 성능을 절약할 수 있습니다. 하위 요소가 많아도 상위 요소에 한 번만 이벤트 리스너를 등록하면 되기 때문입니다.
- 동적으로 추가되거나 삭제되는 하위 요소에도 이벤트가 적용됩니다. 상위 요소에서 버블링된 이벤트를 감지하기 때문입니다.

이벤트 위임은 부모 요소에 하나의 이벤트 핸들러를 추가하여 자식 요소에 여러 이벤트 핸들러를 등록할 필요가 없게 하는 JavaScript 기법입니다. 예를 들어, 다음과 같은 HTML 코드가 있다고 가정해 보세요.

```
<ul id="menu">
  <li data-action="save">Save</li>
  <li data-action="load">Load</li>
  <li data-action="search">Search</li>
</ul>
```

이 코드에서는 각 `li` 요소에 `data-action` 속성을 사용하여 특정 행동을 지정했습니다. 이제 우리는 이벤트 위임을 사용하여 ul 요소에 클릭 이벤트 핸들러를 추가하고, 클릭된 `li` 요소의 `data-action` 속성 값을 가져와서 해당하는 행동을 수행할 수 있습니다. 다음과 같은 JavaScript 코드를 보세요.

```
let menu = document.getElementById('menu');

menu.addEventListener('click', function(event) {
  let target = event.target; // 클릭된 요소
  let action = target.dataset.action; // data-action 속성 값
  if (action) {
    // action 값이 있으면 해당하는 행동 수행
    switch (action) {
      case 'save':
        // 저장 로직
        break;
      case 'load':
        // 불러오기 로직
        break;
      case 'search':
        // 검색 로직
        break;
    }
  }
});
```

이렇게 하면 우리는 각 `li` 요소에 개별적으로 이벤트 핸들러를 등록하지 않아도 되고, 새로운 `li` 요소를 추가해도 자동으로 동일한 로직이 적용됩니다.