삼육대학교 SW융합교육원 김진호

자바스크립트 기초교육

<class>
</body>
<head>
<title>

1-4. DOM (노드 수정)

1-4-1. innerHTML

1-4-2. insertAdjacentHTML

1-4-3. node create append

1-4-4. node insert move

1-4-5. node copy replace remove

1. DOM

1-4. DOM (노드 수정)

1-4-1. innerHTML

- Element.prototype.innerHTML 프로퍼티는 setter와 getter 모두 존재하는 접근자 프로퍼티로 요소 노드의 HTML 마크업을 취득하거나 변경한다.
- textContent 프로퍼티는 HTML 마크업을 무시하고 텍스트만 반환하지만 innerHTML 프로퍼티는 HTML 마크업이 포함 된 문자열을 그대로 반환한다.

```
const $area = document.getElementById('area');

// 읽어온 요소가 가지는 값 출력
console.log($area.innerHTML);

// 노드 추가
$area.innerHTML += '값 추가';

// 노드 교체
$area.innerHTML = "<h1>innerHTML</h1>속성으로 값 변경";

// 노드 삭제
$area.innerHTML = '';
```

```
<div id="area">태그 엘리먼트의 값을 읽거나, 변경할 때
<span>innerHTML</span>속성을 사용함</div>
```

1. DOM

1-4. DOM (노드 수정)

1-4-1. innerHTML

- innerHTML 프로퍼티를 사용한 DOM 조작은 구현이 간단하고 직관적이라는 장점이 있다.
- 하지만 사용자로부터 입력받은 데이터를 그대로 innerHTML 프로퍼티에 할당하는 것은 XSS(Cross-Site Scripting Attacks)에 취약하므로 위험하다는 단점도 있다.
  HTML 마크업 내에 자바스크립트 악성 코드가 포함 되어 있다면 파싱 과정에서 그대로 실행될 가능성이 있다.

```
// 에러 이벤트를 강제로 발생시켜서 자바스크립트 코드가 실행 되도록 한다.
$area.innerHTML = `<img src='x' onerror='alert("메롱메롱~")'>`;
```

1. DOM

1-4. DOM (노드 수정)

1-4-1. innerHTML

- innerHTML 프로퍼티에 HTML 마크업 문자열을 할당하는 경우 요소 노드의 모든 자식 노드를 제거하고 할당한 HTML 마크업 문자열을 파싱 하여 DOM을 변경한다는 단점이 있다.
- 또한, 새로운 요소를 삽입할 때 삽입될 위치를 지정할 수 없다는 단점도 있다.

```
<ul id="list">
    <li class="coffee">커피</li>
</ul>
```

```
const $list = document.getElementById('list');

// 노드 추가
$list.innerHTML += "<li class='coke'>콜라</li>";
// 위 코드는 아래 코드의 축약 표현으로 유지되어도 되는 기존의 자식 노드까지
모두 제거하고 새롭게 자식 노드를 생성하여 DOM에 반영하는 비효율적인 동작을 한다.
// $list.innerHTML = $list.innerHTML + "<li class='coke'>콜라</li>";

// 커피와 콜라 사이에 새로운 요소를 삽입하고 싶을 경우 innerHTML을 통해서는
삽입 위치를 지정할 수 없다.
// 기존 요소를 제거하지 않으면서 위치를 지정해 새로운 요소를 삽입해야 할 때는
insertAdjacentHTML 메서드를 사용하는 것이 더 효율적이다.
```

1. DOM

1-4. DOM (노드 수정)

1-4-2. insertAdjacentHTML

- Element.prototype.insertAdjacentHTML(position, DOMString)
  메서드는 기존 요소를 제거하지 않으면서 위치를 지정해 새로운 요소를 삽입한다.
- 첫 번째 인수로 전달할 수 있는 문자열은 'beforebegin', 'afterbegin',
  'beforeend', 'afterend'의 4가지이다.
- insertAdjacentHTML 메서드는 기존 요소에 영향을 주지 않고 새롭게 삽입될 요소만을 파싱 하여 자식 요소로 추가하므로 기존의 자식 노드를 모드 제거하고 다시 처음부터 새롭게 자식 노드를 생성하여 자식 요소로 추가하는 innerHTML 프로퍼티보다 효율적이고 빠르다. 단, HTML 마크업 문자열을 파싱 하므로 크로스 사이트 스크립팅 공격에 취약하다는 점은 동일하다.

1. DOM

1-4. DOM (노드 수정)

1-4-2. insertAdjacentHTML

```
<div id="area">
    insertAdjacentHTML 메소드 사용 테스트
</div>
```

```
const $area = document.getElementById('area');

$area.insertAdjacentHTML('beforebegin', '<h1>beforebegin 테스트</h1>');
$area.insertAdjacentHTML('afterbegin', '<h1>afterbegin 테스트</h1>');
$area.insertAdjacentHTML('beforeend', '<h1>beforeend 테스트</h1>');
$area.insertAdjacentHTML('afterend', '<h1>afterend 테스트</h1>');
```

1. DOM

1-4. DOM (노드 수정)

1-4-3. node create append

- **Document.prototype.createElement(tagName)**
  : 인수로 전달받은 태그 이름에 해당하는 요소 노드를 생성하여 반환
- **Document.prototype.createTextNode(text)**
  : 인수로 전달받은 텍스트 값으로 텍스트 노드를 생성하여 반환
- **Node.prototype.appendChild(childNode)**
  : 인수로 전달받은 노드를 appendChild 메서드를 호출한 노드의 마지막 자식 노드로 추가

1. DOM

1-4. DOM (노드 수정)

1-4-3. node create append

```
const $drink = document.getElementById('drink');

// 요소 노드 생성
// 기존 DOM에 추가되지 않고 홀로 존재하는 상태, 자식 노드 없는 상태
const $li = document.createElement('li');

// 텍스트 노드 생성
// const textNode = document.createTextNode('콜라');

// 텍스트 노드를 $li 요소 노드의 자식 노드로 추가
// $li.appendChild(textNode);

// 자식 노드가 하나도 없는 경우에는 텍스트 노드를 생성하여
// 요소 노드의 자식 노드로 텍스트 노드를 추가하는 것보다 textContent 사용이 간편
$li.textContent = '콜라';

// $li 요소 노드를 $drink 요소 노드의 마지막 자식 노드로 추가
$drink.appendChild($li);
```

```
<ul id="drink">
    <li>커피</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-3. node create append

- 복수의 노드를 생성하여 추가하는 상황에서 DOM의 리플로우, 리페인트 횟수를 줄이는 것이 좋다.
  - 컨테이너 요소를 사용할 수 있다.
  - DocumentFragment 노드는 자식 노드들의 부모 노드로서
    별도의 서브 DOM을 구성하여 기존 DOM에 추가하기 위한 용도로 사용할 수 있다.

1. DOM

1-4. DOM (노드 수정)

1-4-3. node create append

```
const $food = document.getElementById('food');

// 2) 컨테이너 요소 div 사용
// const $container = document.createElement('div');

// 3) DocumentFragment 노드는 자식 노드들의 부모 노드로서
// 별도의 서브 DOM을 구성하여 기존 DOM에 추가하기 위한 용도로 사용
const $fragment = document.createDocumentFragment();

['된장찌개', '고등어구이', '순대국'].forEach(text => {
    const $li = document.createElement('li');
    $li.textContent = text;

    // 1) DOM이 3번 변경 되면서 리플로우, 리페인트가 3번 실행 되어 비효율적
    // $food.appendChild($li);

    // 2) div에 li를 추가
    // $container.append($li);

    // 3) DocumentFragment 노드에 li를 추가
    $fragment.appendChild($li);
});

// 2) 컨테이너 요소 div를 사용하면 DOM은 1번만 변경
// 하지만 불필요한 요소(div)가 DOM에 추가
// $food.appendChild($container);

// 3) DocumentFragment 노드를 DOM에 추가하면 자신은 제거되고 자식 노드만 추가
// 리플로우와 리페인트도 한 번만 실행
$food.appendChild($fragment);
```

```
<ul id="food">
    <li>김치찌개</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-4. node insert move (노드 삽입과 이동)

- **Node.prototype.appendChild(childNode)**
  : 인수로 전달받은 노드를 appendChild 메서드를 호출한 노드의 마지막 자식 노드로 추가
- **Node.prototype.insertBefore(newNode, childNode)**
  : 첫 번째 인수로 전달받은 노드를 두 번째 인수로 전달 받은 노드 앞에 삽입
  - 두 번째 인수로 전달 받은 노드는 반드시 insertBefore 메서드를 호출한 노드의 자식 노드여야 한다.

1. DOM

1-4. DOM (노드 수정)

1-4-4. node insert move (노드 삽입과 이동)

```
const $drink = document.getElementById('drink');
const $li1 = document.createElement('li');
$li1.textContent = '콜라';

// $li1을 $drink의 마지막 자식 노드로 추가
$drink.appendChild($li1);

const $li2 = document.createElement('li');
$li2.textContent = '우유';

// $li2를 $drink.lastElementChild 앞에 삽입
$drink.insertBefore($li2, $drink.lastElementChild);

const $li3 = document.createElement('li');
$li3.textContent = '사이다';

// 두 번째 인수가 $drink의 자식 노드가 아니여서 DOMException 발생
// $drink.insertBefore($li3, document.querySelector('pre'));

// 두 번째 인수가 null이면 appendChild처럼 마지막 자식 노드로 추가
$drink.insertBefore($li3, null);
```

```
<ul id="drink">
    <li>커피</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-4. node insert move (노드 삽입과 이동)

- DOM에 이미 존재하는 노드를 appendChild, insertBefore 메서드를 사용하여 DOM에 다시 추가하면 현재 위치에서 노드를 제거하고 새로운 위치에 노드를 추가하는 노드 이동이 발생한다.

```
const $food = document.getElementById('food');

// food 하위의 파스타, 피자 노드 취득
const [$pasta, $pizza] = $food.children;
// const $pasta = $food.firstElementChild;
// const $pizza = $food.lastElementChild;

// 위의 ul인 $drink에 마지막 자식 노드로 파스타 추가
$drink.appendChild($pasta);

// 위의 ul인 $drink에 파스타 앞에 피자 추가
$drink.insertBefore($pizza, $pasta);
```

```
<ul id="food">
    <li>파스타</li>
    <li>피자</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-5. node copy replace remove (노드 복사, 교체, 삭제)

- Node.prototype.cloneNode([deep : true | false])
  메서드는 노드의 사본을 생성하여 반환한다.
  - 매개변수 deep에 true를 인수로 전달하면 노드를 깊은 복사하여 모든 자손 노드가 포함된 사본을 생성하고, false를 인수로 전달하거나 생략하면 노드를 얕은 복사하여 노드 자신만의 사본을 생성한다.
  - 얕은 복사로 생성된 요소 노드는 자손 노드를 복사하지 않으므로 텍스트 노드도 없다.

1. DOM

1-4. DOM (노드 수정)

1-4-5. node copy replace remove (노드 복사, 교체, 삭제)

```
const $ul = document.querySelector(".copy");
const $li = $ul.firstElementChild;

// $li 얕은 복사 - 텍스트 노드 없는 사본
const $shallowClone = $li.cloneNode(false);
// $ul 깊은 복사 - 자손 노드 li, 텍스트 노드가 있는 사본
const $deepClone = $ul.cloneNode(true);

// $ul의 마지막 노드로 추가
$ul.appendChild($shallowClone);
$ul.appendChild($deepClone);
```

```
<ul class="copy">
    ul 영역
    <li>li 영역</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-5. node copy replace remove (노드 복사, 교체, 삭제)

- Node.prototype.replaceChild(newChild, oldChild) 메서드는 자신을 호출한 노드의 자식 노드인 oldChild 노드를 newChild 노드로 교체한다.
- 이때 oldChild는 replaceChild 메서드를 호출한 노드의 자식 노드여야 하고 oldChild 노드는 DOM에서 제거된다.

```
const $drink = document.getElementById('drink');
const $coffee = $drink.firstElementChild;

const $newChild = document.createElement('li');
$newChild.textContent = '콜라';

// 커피 요소 노드를 $newChild 요소 노드로 교체
$drink.replaceChild($newChild, $coffee);
```

```
<ul id="drink">
    <li>커피</li>
    <li>사이다</li>
</ul>
```

1. DOM

1-4. DOM (노드 수정)

1-4-5. node copy replace remove (노드 복사, 교체, 삭제)

- Node.prototype.removeChild(child) 메서드는 child 매개변수에 인수로 전달한 노드를 DOM에서 삭제한다.
- 인수로 전달한 노드는 removeChild 메서드를 호출한 노드의 자식이어야 한다.

```
<ul id="food">
    <li>파스타</li>
    <li>피자</li>
</ul>
```

```
const $food = document.getElementById('food');

// $food 요소 노드의 마지막 요소를 DOM에서 삭제
$food.removeChild($food.lastElementChild);
```