12. HTML과 JavaScript 연동

12.1 `<script>` 태그 사용 위치

— HTML 문서에서 JavaScript를 포함할 때의 위치에 따른 차이와 최적화 전략

HTML에서 JavaScript 코드를 포함하기 위한 `<script>` 태그는
문서 내에서 **어디에 위치하느냐에 따라 웹페이지의 로딩 속도, 동작 시점, 사용자 경험**에 큰 영향을 미친다.

이 항목에서는 `<script>` 태그의 주요 사용 위치와 각 위치에 따른 특징, 실전 권장 방식까지 정리한다.

✅ 1. `<head>` 내부에 위치하는 경우

```
<!DOCTYPE html>
<html>
<head>
  <script src="script.js"></script>
</head>
<body>
  <h1>Hello</h1>
</body>
</html>
```

📌 **특징**:

- HTML을 렌더링하기 **전에** JavaScript가 먼저 로드됨
- **페이지 렌더링이 차단됨**
    - JS가 로딩/해석/실행되기 전까지 HTML 렌더링이 멈춤
- 외부 라이브러리 필수 로딩 시 종종 사용됨 (예: jQuery 등)

👄 **단점**:

- 초기 페이지 로딩 속도 저하
- DOM 요소를 조작하는 스크립트가 `null` 오류를 발생시킬 수 있음

✅ 2. `<body>` 끝에 위치하는 경우 (권장)

```
<!DOCTYPE html>
<html>
<head>
  <title>Document</title>
</head>
<body>
  <h1>Hello</h1>
  <script src="script.js"></script>
</body>
</html>
```

📌 **특징**:

- HTML 요소가 **전부 렌더링된 이후에** JS 로드됨
- 페이지 로딩 차단 없음
- **DOM 조작 안전** (요소들이 이미 존재하므로)

✅ **가장 일반적이고 권장되는 위치**
→ JS가 사용자 인터페이스 요소에 접근할 수 있음

✅ 3. `<script defer>` 사용 (`<head>` 내부 가능)

```
<head>
  <script src="script.js" defer></script>
</head>
```

📌 **특징**:

- HTML 파싱과 **병렬로 스크립트를 다운로드**
- HTML이 **완전히 파싱된 후** JS 실행됨
- `<body>` 끝에 넣는 것과 유사하지만 **더 명시적**

✅ **최신 브라우저 환경에서 선호되는 방식**
→ 특히 여러 스크립트를 사용하는 경우 안정적 실행 보장

✅ 4. `<script async>` 사용

```
<head>
  <script src="analytics.js" async></script>
</head>
```

📌 **특징**:

- **HTML 파싱과 동시에 다운로드 및 실행**
- 실행 시점이 예측 불가능 (먼저 로드된 순으로 실행)

- **비동기 스크립트 실행 (페이지 렌더링과 병행)**

📌 사용 예:

- 광고, 추적기, 통계 분석 등의 **비필수 외부 JS 삽입**

👄 단점:

- 여러 스크립트가 순서 보장 없이 실행됨
- DOM 의존 스크립트에는 부적합

✅ 우선순위 및 실행 타이밍 비교

위치/속성	다운로드 시점	실행 시점	DOM 안전성	권장 사용
`<head>`	HTML 파싱 전	즉시 실행 (파싱 차단)	❌ 위험	초기 로딩 필요 시
`<body>` 끝	HTML 파싱 후	HTML 렌더링 후	✅ 안전	일반 스크립트
`<script defer>`	병렬 다운로드	DOMContentLoaded 후 실행	✅ 안전	모듈형 JS 권장
`<script async>`	병렬 다운로드	다운로드 완료 즉시 실행	❌ 위험	통계/비핵심 JS

✅ 실전 권장 방식

```
<head>
  <!-- 비동기 통계 -->
  <script src="analytics.js" async></script>

  <!-- 필수 JS (DOM 조작 필요) -->
  <script src="main.js" defer></script>
</head>
```

또는...

```
<body>
  <!-- HTML 렌더링 후 실행 -->
  <script src="main.js"></script>
</body>
```

✅ 한 줄 요약

> `<script>` 태그는 가능한 한 `<body>` **끝** 또는 `defer` **속성을 활용**해 DOM 렌더링을 차단하지 않도록 구성하는 것이 현대적인 웹 성능 최적화의 핵심이다.

12.2 `defer`, `async` 속성

— HTML에서 JavaScript를 로드하고 실행하는 방식 제어

HTML `<script>` 태그는 `src` 속성을 통해 외부 JavaScript 파일을 불러올 수 있는데,
이때 **스크립트의 다운로드 및 실행 시점을 제어하는 속성**으로 `defer`와 `async`가 사용된다.
이들은 성능 최적화와 렌더링 방해 방지에 핵심적 역할을 한다.

✅ 기본 동작 (`<script src="...">`만 사용한 경우)

```
<head>
  <script src="script.js"></script>
</head>
```

📌 기본적으로 `<script>` 태그는 **HTML 파싱을 중단**시키고
JS 파일을 다운로드 및 실행한 후, 다시 HTML 파싱을 이어감.
→ 이로 인해 **렌더링 차단(blocking)** 문제가 발생할 수 있음.

✅ `defer` 속성

```
<head>
  <script src="script.js" defer></script>
</head>
```

🔷 특징

항목	설명
다운로드 시점	HTML 파싱 중에 **병렬로 다운로드**
실행 시점	HTML 파싱 완료 후 (`DOMContentLoaded` 직전)
실행 순서	**스크립트 순서 보장됨**
DOM 접근	가능 (모든 요소 파싱 완료 후 실행되므로)

✅ **정상적인 DOM 조작, 복수 파일 실행 순서 보장**에 가장 적합
✅ 여러 `defer` 스크립트는 **정해진 순서대로 실행됨**

📌 예시

```
<script src="lib.js" defer></script>
<script src="main.js" defer></script>
```

→ `lib.js` → `main.js` 순서 보장됨

✅ `async` 속성

```
<head>
  <script src="script.js" async></script>
</head>
```

🔷 특징

항목	설명
다운로드 시점	HTML 파싱 중에 **병렬로 다운로드**
실행 시점	**다운로드 완료 즉시 실행**
실행 순서	**보장되지 않음**
DOM 접근	보장되지 않음 (렌더링 중간에 실행될 수 있음)

✅ **광고, 추적기, 분석기**처럼 DOM 의존이 없고 순서와 무관한 스크립트에 적합
👄 DOM 요소가 아직 생성되지 않은 시점에 실행되면 **오류 발생 가능**

✅ 실행 순서 비교 (시뮬레이션)

✅ 예시 1: 기본 `<script>`

- HTML 파싱 중단 → JS 다운로드 및 실행 → 파싱 재개

✅ 예시 2: `<script defer>`

- JS 병렬 다운로드
- HTML 파싱 완료 후 실행
- **순서 보장**

✅ 예시 3: `<script async>`

- JS 병렬 다운로드
- 다운로드 완료된 즉시 실행
- **순서 비보장**

✅ 요약 표

속성	다운로드	실행 시점	실행 순서 보장	DOM 접근 안전성	주용도
없음	즉시	즉시	O	❌ 위험	레거시 방식, 권장 안 함
`defer`	병렬	HTML 파싱 후	✅ 보장	✅ 안전	기본 스크립트 처리
`async`	병렬	다운로드 완료 후	❌ 비보장	❌ 위험	비동기 외부 스크립트

✅ 실전 권장 구성

```
<!-- DOM 조작 포함된 스크립트 -->
<script src="/js/app.js" defer></script>

<!-- 외부 통계/광고 비동기 삽입 -->
<script src="https://www.google-analytics.com/analytics.js" async></script>
```

✅ 한 줄 요약

> `defer`는 **DOM 파싱 이후 실행 + 순서 보장**,
> `async`는 **즉시 실행 + 순서 무관**,
> 각각의 용도와 실행 시점을 고려해 사용해야 웹 성능과 안정성을 극대화할 수 있다.

12.3 DOM 요소 조작 준비

— JavaScript에서 안전하게 HTML 요소(DOM)를 조작하기 위한 실행 시점 관리

웹 페이지에서 JavaScript가 HTML 요소를 조작하려면
해당 요소가 **이미 DOM에 존재하고 파싱이 완료된 이후**여야 한다.
그렇지 않으면 `null` 반환이나 **오류**가 발생한다.

이 섹션에서는 DOM 요소를 조작할 수 있는 **시점 제어 방법**을
가장 일반적인 3가지 방식으로 설명한다:

✅ 1. `<body>` 끝에 스크립트 위치시키기

```
<body>
  <button id="btn">Click me</button>
  <script>
    const btn = document.getElementById("btn");
    btn.addEventListener("click", () => alert("Hello!"));
  </script>
</body>
```

📌 `<script>`가 **HTML 요소 뒤에 위치**하므로
JS 실행 시점에 이미 DOM이 파싱되어 있고, `btn` 요소를 정상적으로 참조할 수 있음.

✅ **가장 간단하고 직관적인 방식**

✅ 2. `DOMContentLoaded` 이벤트 사용

```
<script>
  document.addEventListener("DOMContentLoaded", () => {
    const btn = document.getElementById("btn");
    btn.addEventListener("click", () => alert("Hello DOM!"));
  });
</script>
```

📌 `DOMContentLoaded` 이벤트는
모든 HTML 요소의 파싱이 끝난 직후 발생하며,
CSS/이미지 로딩은 기다리지 않음.

✅ `<head>`**에 script를 넣더라도 안전하게 DOM을 조작**할 수 있음.

✅ 3. `window.onload` 이벤트 사용

```
<script>
  window.onload = () => {
    const btn = document.getElementById("btn");
    btn.addEventListener("click", () => alert("Everything loaded!"));
  };
</script>
```

📌 `window.onload`는
HTML + CSS + 이미지 등 모든 리소스가 로딩된 이후에 실행됨

⛔ DOM 조작에는 **불필요하게 늦은 시점**이므로 비권장

✅ 모든 외부 리소스까지 준비된 후 실행되어야 하는 경우 (예: 이미지 크기 계산 등)에만 사용

✅ 비교 요약표

방식	실행 시점	DOM 안전성	이미지 등 리소스 대기	권장 여부
`<script>`를 `<body>` 끝에	HTML 요소 파싱 후	✅ 안전	❌ (대기 안 함)	⭐ 매우 권장
`DOMContentLoaded` 이벤트	HTML 파싱 완료 후	✅ 안전	❌	⭐ 매우 권장
`window.onload` 이벤트	HTML + 이미지 등 전부 완료 후	✅ 안전	✅ (대기함)	⚠️ 상황에 따라

✅ 실전 권장 패턴 요약

```
document.addEventListener("DOMContentLoaded", () => {
  // DOM 접근 및 조작 안전
});
```

또는 HTML 말미에 스크립트 배치:

```
<body>
  <!-- 요소들 -->
  <script src="main.js"></script>
</body>
```

✅ 참고: `defer` 속성과의 연계

```
<head>
  <script src="main.js" defer></script>
</head>
```

📌 `defer`를 사용하면 JS는 자동으로 **DOM 파싱이 끝난 후에 실행되므로**,
별도로 `DOMContentLoaded` 이벤트를 걸 필요 없이 DOM 요소를 안전하게 조작할 수 있다.

✅ 한 줄 요약

> DOM 요소를 조작하려면 HTML 파싱이 완료된 후 실행되어야 하며,
> 이를 위해서는 `<body>` 끝에 JS 배치하거나 `DOMContentLoaded` 이벤트를 사용하는 것이 가장 안전하고 일반적이다.

12.4 이벤트 속성 (`onclick`, `onchange`, 등)

— HTML 요소에 직접 JavaScript 이벤트를 연결하는 전통적인 방법

HTML은 사용자의 상호작용(클릭, 키보드 입력, 마우스 움직임 등)에 반응하기 위해
이벤트 속성(Event Attributes)을 제공한다.
이 속성은 HTML 태그에 직접 JavaScript 코드를 작성하는 방식이다.

✅ 1. 주요 이벤트 속성 정리

속성	설명	예시 요소
`onclick`	클릭 이벤트	버튼, 링크
`ondblclick`	더블 클릭 이벤트	div, 버튼
`onchange`	값이 변경될 때 (입력, 선택 등)	input, select

속성	설명	예시 요소
`oninput`	값이 입력되는 순간마다	input, textarea
`onfocus`	요소가 포커스를 얻을 때	input, textarea
`onblur`	요소가 포커스를 잃을 때	input
`onkeydown`	키가 눌릴 때	input, 문서
`onkeyup`	키가 떼어질 때	input, 문서
`onmouseover`	마우스가 요소 위로 올려질 때	div, span
`onmouseout`	마우스가 요소를 벗어날 때	div, span
`onsubmit`	폼 제출 시	form
`onreset`	폼 리셋 시	form

✅ 2. 기본 사용법 (HTML 내 직접 작성)

```
<button onclick="alert('버튼이 클릭되었습니다!')">클릭</button>
```

📌 클릭 시 `alert()` 함수 실행
👄 **HTML과 JS가 분리되지 않아 유지보수성이 낮음**

✅ 3. 여러 이벤트 예제

```
<input type="text" onchange="alert('값이 변경되었습니다')">
<select onchange="console.log(this.value)">
  <option value="one">One</option>
  <option value="two">Two</option>
</select>

<form onsubmit="alert('제출!'); return false;">
  <button type="submit">Submit</button>
</form>
```

📌 `return false`는 폼의 기본 제출을 막음

✅ 4. JS로 연결하는 방식과의 차이점

구분	이벤트 속성 (`onclick`)	JS로 연결 (`addEventListener`)
위치	HTML 태그 내부	JS 코드에서 분리됨
복수 이벤트 등록	❌ 불가능 (1개만 가능)	✅ 가능

구분	이벤트 속성 (`onclick`)	JS로 연결 (`addEventListener`)
유지보수	❌ HTML + JS 혼합	✅ 역할 분리
표준성	과거 방식 (지원은 됨)	현대 표준 방식

✅ 권장 방식 (addEventListener)

```
<button id="btn">클릭</button>
<script>
  document.getElementById("btn").addEventListener("click", () => {
    alert("이벤트 리스너 방식!");
  });
</script>
```

📌 역할을 분리하고, 여러 이벤트도 유연하게 다룰 수 있음

✅ 언제 이벤트 속성을 쓸까?

상황	권장 여부
빠르게 테스트할 때	가능
학습/데모용으로 간단히 구현 시	가능
실무 웹사이트	❌ 비권장

✅ 실전 예시 모음

```
<!-- oninput -->
<input type="text" oninput="console.log(this.value)">

<!-- onkeydown -->
<input type="text" onkeydown="if(event.key === 'Enter') alert('엔터!')">

<!-- onmouseover / onmouseout -->
<div onmouseover="this.style.background='lightblue'"
     onmouseout="this.style.background=''">
  마우스를 올려보세요
</div>
```

✅ 한 줄 요약

> `onclick`, `onchange` 등의 이벤트 속성은 **HTML 요소에 직접 이벤트 처리기를 지정하는 방법**이며,
> **단순한 구현에는 유용하지만, 실무에서는 `addEventListener()` 기반의 분리된 이벤트 바인딩이 표준이다.**