DOM Manipulation mit JavaScript

Kapitel 1: Einführung in das DOM (Document Object Model)

Was ist das DOM?

Das **Document Object Model (DOM)** ist eine standardisierte, objektorientierte API, mit der man **strukturierte Dokumente** wie HTML oder XML über Programmiersprachen wie JavaScript **lesen, manipulieren und dynamisch verändern** kann. Es stellt die Struktur eines Dokuments als **hierarchischen Baum** dar, wobei jeder Teil des Dokuments ein **Knoten (Node)** ist:

- Jedes **HTML-Element** (wie `<div>`, `<p>`, `<a>`, etc.) ist ein **Elementknoten**.
- Jedes **Attribut** eines Elements ist ein **Attributknoten**.
- Der **Text** zwischen Elementen ist ein **Textknoten**.

Der Einstiegspunkt für alle DOM-Operationen ist das globale `document`-Objekt.

Analogie

Stell dir das DOM als einen Baum vor:

- Der **Wurzelknoten** ist `document`.
- **Zweige** sind HTML-Elemente.
- **Blätter** sind Texte oder Attribute.

Die Struktur des DOM-Baums

Wenn eine HTML-Datei geladen wird, wird sie vom Browser geparst und als DOM-Baum dargestellt. Dieser Baum besteht aus verschiedenen Knoten, die miteinander verwandt sind:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Einfaches DOM Beispiel</title>
  </head>
  <body>
    <h1>Willkommen</h1>
    <p>Dies ist ein Absatz.</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
    </ul>
  </body>
</html>
```

Diese HTML-Datei wird vom DOM wie folgt strukturiert:

```
document
├── html
│   ├── head
│   │   └── title (Text: "Einfaches DOM Beispiel")
│   └── body
│       ├── h1 (Text: "Willkommen")
│       ├── p (Text: "Dies ist ein Absatz.")
│       └── ul
│           ├── li (Text: "Item 1")
│           └── li (Text: "Item 2")
```

Jedes Element hat Eltern- und Kindbeziehungen, was uns erlaubt, durch den Baum zu navigieren.

DOM vs HTML

HTML	DOM
Statischer Text	Dynamische Struktur im Speicher
Geschrieben vom Entwickler	Generiert durch den Browser
Veränderbar nur durch Reload	Kann durch JavaScript modifiziert werden

Bedeutung für die Webentwicklung

Die DOM-Manipulation ist eine **Kernkompetenz** für moderne Webentwicklung, weil sie es ermöglicht:

- Inhalte dynamisch zu ändern (z.B. nach API-Calls)
- Benutzerinteraktionen zu verarbeiten (Formular-Validierung, Animationen, etc.)
- Inhalte je nach Kontext (z.B. Sprache, Uhrzeit, Nutzerrolle) anzupassen

Erste DOM-Operationen mit JavaScript

Beispiel: Zugriff auf ein Element und Ausgabe in der Konsole:

```
let header = document.querySelector("h1");
console.log(header.textContent); // Ausgabe: "Willkommen"
```

Elementinhalt manipulieren:

```
header.textContent = "DOM-Manipulation ist cool!";
```

Neues Element hinzufügen:

```
let newItem = document.createElement("li");
newItem.textContent = "Item 3";
document.querySelector("ul").appendChild(newItem);
```

Kapitel 2: Zugriff auf DOM-Elemente

Warum DOM-Elemente gezielt auswählen?

Bevor man Elemente verändern oder Ereignisse daran binden kann, muss man sie **aus dem DOM selektieren**. Dafür stellt das `document`-Objekt verschiedene Methoden zur Verfügung. Diese Methoden liefern entweder einzelne Elemente oder Sammlungen zurück.

Selektoren im Überblick

`document.getElementById()`

- Liefert **ein einzelnes Element** mit einer bestimmten `id`
- Schnell, da `id` einzigartig ist

```
let header = document.getElementById("main-title");
```

`document.querySelector()`

- Gibt das **erste Element** zurück, das auf einen **CSS-Selektor** passt
- Sehr flexibel

```
let firstParagraph = document.querySelector("p");
let mainHeader = document.querySelector("#main-title");
let button = document.querySelector(".btn");
```

`document.querySelectorAll()`

- Gibt **alle Elemente** zurück, die auf den CSS-Selektor passen
- Rückgabe ist eine **NodeList** (nicht live!)

```
let paragraphs = document.querySelectorAll("p");
paragraphs.forEach(p => console.log(p.textContent));
```

`document.getElementsByClassName()`

- Gibt eine **HTMLCollection** aller Elemente mit einer bestimmten Klasse zurück
- Live Collection

```
let boxes = document.getElementsByClassName("box");
for (let box of boxes) {
    console.log(box.textContent);
}
```

`document.getElementsByTagName()`

- Gibt alle Elemente eines bestimmten Typs zurück, z. B. alle `<li>`-Elemente

```
let listItems = document.getElementsByTagName("li");
```

Vergleich: NodeList vs. HTMLCollection

Eigenschaft	**NodeList (`querySelectorAll`)**	**HTMLCollection (`getElementsByClassName`)**
Live-Update	Nein	Ja
Iterierbarkeit	Ja (forEach etc.)	Ja (for..of, aber nicht forEach direkt)
Rückgabetyp	Statische Liste	Dynamisch verknüpfte Liste

Performance & Best Practices

- `getElementById()` ist am schnellsten, aber auf `id`s beschränkt
- `querySelector()` ist flexibel und modern, ideal für einzelne Elemente
- Verwende `querySelectorAll()` für viele Elemente, aber beachte: **keine Live-Collection**
- Wenn du mit sich dynamisch ändernden DOM-Elementen arbeitest, bevorzuge Live-Collections (`getElementsByClassName`)

Selektoren kombinieren

CSS-Selektoren können kombiniert werden, um präzise Abfragen zu machen:

```
let selected = document.querySelector(".card > .title.highlight");
```

Kapitel 2b: Live vs. Static NodeLists

Beim Zugriff auf DOM-Elemente kann JavaScript entweder eine **statische Liste** oder eine **dynamische ("live") Sammlung** zurückgeben. Der Unterschied ist wichtig, wenn sich der DOM während der Laufzeit verändert – etwa durch das Erstellen oder Entfernen von Elementen.

---

1. Was ist eine NodeList?

Eine **NodeList** ist eine Sammlung von DOM-Knoten. Sie entsteht z. B. durch:

- `document.querySelectorAll()` → gibt eine **statische** NodeList zurück
- `childNodes` → gibt in der Regel eine **live** NodeList zurück (Ausnahme!)

Eigenschaften:

- Kann mit `forEach()` durchlaufen werden
- Hat eine `length`
- Wird **nicht automatisch aktualisiert**, wenn sich das DOM verändert

```
let items = document.querySelectorAll(".item");
console.log(items.length); // z. B. 3
```

---

2. Was ist eine HTMLCollection?

Eine **HTMLCollection** ist eine **live Collection** – das heißt, sie ändert sich automatisch, wenn sich das DOM ändert. Sie entsteht durch:

- `getElementsByClassName()`
- `getElementsByTagName()`
- `children`

Eigenschaften:

- Kein `forEach()` direkt – muss in Array umgewandelt werden oder via `for..of` durchlaufen
- Wird **automatisch aktualisiert**, wenn sich die Struktur des DOM ändert

```
let items = document.getElementsByClassName("item");
let newItem = document.createElement("div");
newItem.className = "item";
document.body.appendChild(newItem);
console.log(items.length); // erhöht sich direkt
```

---

3. Direktvergleich: NodeList vs. HTMLCollection

Eigenschaft	NodeList	HTMLCollection
Iterierbar mit `forEach()`	✅ Ja	❌ Nein (nur `for..of`)
Automatisch aktualisiert?	❌ Nein	✅ Ja
Rückgabetyp von	`querySelectorAll()`	`getElementsByClassName()`
Typ	`NodeList`	`HTMLCollection`

---

4. Live vs. Static in der Praxis

Fallbeispiel:

```
let liveItems = document.getElementsByClassName("item");
let staticItems = document.querySelectorAll(".item");

let newItem = document.createElement("div");
newItem.className = "item";
document.body.appendChild(newItem);

console.log("Live:", liveItems.length);   // Wird erhöht
console.log("Static:", staticItems.length); // Bleibt gleich
```

---

5. Best Practices

- Verwende `querySelectorAll()` für einmalige, gezielte Abfragen und wenn du `forEach()` direkt nutzen willst
- Verwende `getElementsByClassName()` oder `children`, wenn du eine dynamisch aktualisierte Liste brauchst
- Konvertiere HTMLCollections wenn nötig:

```
Array.from(items).forEach(el => ...);
```

Kapitel 3: DOM-Knoten und Navigation

DOM-Knotenarten

Der DOM-Baum besteht aus verschiedenen **Knotenarten (Node Types)**. Die wichtigsten sind:

- **Element-Knoten** (`nodeType === 1`): HTML-Elemente wie `<div>`, `<p>` usw.
- **Text-Knoten** (`nodeType === 3`): Enthalten Text zwischen den Tags
- **Kommentar-Knoten** (`nodeType === 8`): Inhalte zwischen `<!-- -->`
- **Dokument-Knoten** (`document`): Repräsentiert das gesamte HTML-Dokument

Beispiel:

```
<div>
  <p>Textinhalt</p>
</div>
```

- `<div>` ist ein Elementknoten
- `<p>` ist ein Elementknoten
- "Textinhalt" ist ein Textknoten

Navigation durch den DOM-Baum

Um DOM-Elemente zu traversieren, stellt das DOM eine Reihe von Navigations-Eigenschaften zur Verfügung:

Eigenschaft	Beschreibung
`parentNode`	Gibt das Elternelement eines Knotens zurück
`childNodes`	Gibt eine NodeList aller direkten Kinderknoten zurück
`children`	Gibt eine HTMLCollection aller Kind-**Elemente** zurück
`firstChild` / `lastChild`	Gibt das erste / letzte Kind (inkl. Textknoten) zurück
`firstElementChild` / `lastElementChild`	Gibt erstes/letztes **Element** zurück
`nextSibling` / `previousSibling`	Gibt das nächste/vorherige Geschwister-**Node** zurück
`nextElementSibling` / `previousElementSibling`	Gibt das nächste/vorherige **Element** zurück

Beispiel zur Navigation

```
<div id="container">
  <h1>Überschrift</h1>
  <p>Erster Absatz</p>
  <p>Zweiter Absatz</p>
</div>
```

```
const container = document.getElementById("container");
console.log(container.childNodes); // NodeList mit Text- und Elementknoten
console.log(container.children);   // HTMLCollection nur mit <h1> und <p>

const firstPara = container.children[1];
console.log(firstPara.previousElementSibling.textContent); // Gibt "Überschrift" zurück
```

Besonderheiten bei `childNodes`

Die `childNodes`-Liste enthält auch **Textknoten**, z. B. für Leerzeichen und Zeilenumbrüche. Das kann zu unerwartetem Verhalten führen:

```
console.log(container.childNodes.length); // Kann größer sein als container.children.length
```

DOM Traversal Best Practices

- Verwende `children` statt `childNodes`, wenn du nur **Elemente** brauchst.
- Nutze `element.querySelector(...)` für gezielte Navigation statt Traversal über mehrere Ebenen.
- Nutze `closest(selector)` um aufwärts im DOM zu traversieren.

```
let button = document.querySelector("button");
let form = button.closest("form");
```

Kapitel 4: Lesen und Schreiben von Inhalten im DOM

Übersicht

In diesem Kapitel lernen wir, wie man mit JavaScript auf den **Inhalt** von DOM-Elementen zugreift und diesen verändert. Dabei gibt es unterschiedliche Methoden, je nachdem ob man reinen Text, HTML oder bestimmte Eigenschaften eines Elements ansprechen möchte.

---

1. `innerText`, `textContent` und `innerHTML`

`innerText`

- Gibt den **sichtbaren Text** eines Elements zurück (wie er im Browser erscheint)
- Ignoriert unsichtbaren Inhalt (z. B. via `display: none`)
- Löst ein **Reflow/Repaint** im Browser aus (langsamer)

```
let heading = document.querySelector("h1");
console.log(heading.innerText);
```

`textContent`

- Gibt den **vollständigen Textinhalt** eines Elements zurück (inkl. unsichtbarer Zeichen)

- Schnell, da kein Layout neu berechnet wird

```
console.log(heading.textContent);
heading.textContent = "Neuer Titel";
```

`innerHTML`

- Gibt den **HTML-Inhalt** eines Elements als String zurück
- Kann auch zum **Einfügen von HTML** genutzt werden

```
let box = document.querySelector(".box");
box.innerHTML = "<strong>Fetter Text</strong>";
```

> **Hinweis**: `innerHTML` ist sehr mächtig, birgt aber auch **Sicherheitsrisiken (XSS)**, wenn HTML aus Benutzereingaben eingefügt wird.

---

2. Weitere Eigenschaften: `.value`, `.src`, `.href`, `.alt`, `.className`, `.id`

Diese Eigenschaften beziehen sich auf spezifische Arten von DOM-Elementen:

`.value`

- Für Formulareingaben (Input, Textarea, Select)

```
let input = document.querySelector("input");
input.value = "Benutzername";
```

`.src` / `.href`

- Bildquellen bzw. Linkziele

```
let image = document.querySelector("img");
image.src = "neues-bild.jpg";

let link = document.querySelector("a");
link.href = "https://example.com";
```

`.alt`, `.className`, `.id`

```
image.alt = "Alternativer Bildtext";
link.className = "wichtig";
link.id = "start-link";
```

---

3. Performance und Sicherheit: Welche Methode wann?

Eigenschaft	Enthält HTML?	Sichtbarkeit beachtet?	Performance	Sicherheit
`innerText`	Nein	Ja	Langsamer	Sicher
`textContent`	Nein	Nein	Schnell	Sicher
`innerHTML`	Ja	Nein	OK	**Gefährlich bei User-Input!**

Empfehlung:

- Verwende `textContent` für normalen Text
- Verwende `innerHTML` nur für HTML, niemals mit **ungeprüftem Input**
- `innerText` nur wenn du auf sichtbaren Text abzielst

Kapitel 5: Attribute lesen und setzen

Was sind HTML-Attribute?

Attribute liefern **zusätzliche Informationen** über HTML-Elemente. Beispiele sind `href` bei Links, `src` bei Bildern oder `type` bei Input-Feldern. In JavaScript lassen sich diese Attribute **lesen, setzen, ändern oder entfernen**.

---

1. `.getAttribute()` – Attribut lesen

```
let link = document.querySelector("a");
console.log(link.getAttribute("href")); // z.B. "https://example.com"
```

- Gibt den **ursprünglichen Attributwert** zurück (auch wenn z. B. `.href` eine absolute URL liefern würde)

---

2. `.setAttribute()` – Attribut setzen oder überschreiben

```
link.setAttribute("target", "_blank");
link.setAttribute("title", "Öffnet in neuem Tab");
```

- Setzt ein Attribut oder **überschreibt** es, wenn es bereits existiert

---

3. `.removeAttribute()` – Attribut entfernen

```
link.removeAttribute("target");
```

- Entfernt das Attribut vollständig vom Element

---

4. Eigenschaften vs. Attribute (Property vs. Attribute)

Es gibt Unterschiede zwischen **DOM-Eigenschaften** und **HTML-Attributen**:

Zugriffsmethode	Typ	Beispiel
`element.href`	Eigenschaft (Property)	Vollständig aufgelöste URL
`element.getAttribute("href")`	Attribut	Originalwert aus HTML

Beispiel:

```
<a href="/seite.html">Link</a>
```

```
let link = document.querySelector("a");
console.log(link.href);              // https://www.deine-seite.de/seite.html
console.log(link.getAttribute("href")); // /seite.html
```

> **Merksatz**: Eigenschaften geben oft **verarbeitete Daten** zurück, Attribute den **Originalwert aus dem HTML**.

---

5. Datensätze mit `data-*`-Attributen

HTML erlaubt das Setzen eigener Datenfelder mittels `data-`-Attributen. Diese sind über `.dataset` in JS zugänglich.

```
<div id="user" data-user-id="42" data-role="admin"></div>
```

```
let userDiv = document.getElementById("user");
console.log(userDiv.dataset.userId); // "42"
console.log(userDiv.dataset.role);   // "admin"
```

---

6. Typische Anwendungsfälle

- Tooltips mit `title`
- Links und Bildpfade dynamisch setzen
- Klassenbasiertes Verhalten durch `data-*` definieren
- Bedingte Darstellung durch Attributpräsenz

Kapitel 5b: DOM-Properties im Überblick – nach Elementtyp

Warum Properties?

DOM-Elemente in JavaScript besitzen **Eigenschaften (Properties)**, die direkt auf sie bezogen sind. Diese unterscheiden sich teilweise von HTML-Attributen und bieten direkten Zugriff auf den aktuellen Zustand eines Elements – z. B. ob eine Checkbox aktiv ist oder welche Option ausgewählt wurde.

> **Wichtig:** Nicht verwechseln mit `getAttribute()` – Properties sind live und spiegeln den Zustand zur Laufzeit wider.

---

1. Allgemein gängige Properties (für fast alle Elemente)

Property	Beschreibung
`.id`	Die ID des Elements
`.className`	Klassen als String
`.classList`	Zugriff auf Klassen als Liste (empfohlen)
`.style`	Zugriff auf Inline-Stile
`.title`	Tooltip-Text
`.hidden`	Boolean: Ob das Element versteckt ist

---

2. `<input>` und `<textarea>`

Property	Beschreibung
`.value`	Der eingegebene Textinhalt
`.checked`	Ob ein Checkbox/Radio ausgewählt ist
`.disabled`	Ob das Feld deaktiviert ist
`.type`	Typ des Eingabefelds (text, checkbox...)
`.placeholder`	Platzhaltertext

Property	Beschreibung
`.name`	Name-Attribut

Beispiel:

```
let input = document.querySelector("input");
input.value = "Max Mustermann";
if (input.checked) { ... }
```

---

3. `<select>` und `<option>`

Property	Beschreibung
`.value`	Wert der ausgewählten Option
`.selectedIndex`	Index der ausgewählten Option
`.options`	HTMLCollection aller Options
`.disabled`	Ob das Feld deaktiviert ist

Beispiel:

```
let select = document.querySelector("select");
console.log(select.value);
console.log(select.options[select.selectedIndex].text);
```

---

4. `<img>` (Bild)

Property	Beschreibung
`.src`	Bildquelle
`.alt`	Alternativtext
`.width`	Breite in Pixeln (veränderbar)
`.height`	Höhe in Pixeln (veränderbar)
`.naturalWidth`	Ursprüngliche Bildbreite
`.naturalHeight`	Ursprüngliche Bildhöhe

---

5. `<a>` (Link)

Property	Beschreibung
`.href`	Zieladresse
`.target`	Ziel-Fenster (_blank, _self, ...)
`.rel`	Beziehung zum Ziel (noreferrer, ...)
`.textContent`	Linktext

---

6. `<form>`

Property	Beschreibung

Property	Beschreibung
`.action`	Ziel-URL beim Absenden
`.method`	Methode (GET oder POST)
`.elements`	Sammlung aller enthaltenen Formularelemente
`.submit()`	Methode zum programmatischen Absenden

---

7. Wann Property, wann Attribut?

Ziel	Verwende
Aktueller Wert eines Inputs	`.value`, `.checked`, etc.
Originalwert aus HTML	`.getAttribute()`
Dynamische Updates (JS-Zustand)	Property (z. B. `.disabled = true`)

> **Merksatz:** Properties = „aktuell“ / Attribute = „ursprünglich“

Kapitel 6: Klassen- und Stilmanipulation

1. Klassen bearbeiten mit `classList`

Das `classList`-Objekt eines Elements bietet eine bequeme und moderne Möglichkeit, CSS-Klassen zu verwalten, ohne mit kompletten Strings arbeiten zu müssen.

Methoden von `classList`

Methode	Beschreibung
`.add("klasse")`	Fügt die Klasse hinzu (wenn nicht vorhanden)
`.remove("klasse")`	Entfernt die Klasse
`.toggle("klasse")`	Entfernt Klasse, wenn vorhanden, andernfalls fügt sie sie hinzu
`.contains("klasse")`	Gibt `true` zurück, wenn die Klasse vorhanden ist

Beispiel:

```
let box = document.querySelector(".box");
box.classList.add("aktiv");
box.classList.toggle("sichtbar");
```

> **Hinweis**: Vermeide `element.className += " neue-klasse"` – das überschreibt eventuell bestehende Klassen!

---

2. Direktes Zuweisen von Klassen: `className`

```
box.className = "neue-klasse";
```

- Setzt die **komplette Klassenliste neu**
- Gefahr: Bestehende Klassen werden überschrieben

---

3. Inline-Stile bearbeiten mit `style`

Das `style`-Objekt erlaubt es, CSS-Stile direkt auf ein Element anzuwenden. Diese ändern **Inline-Styles**, nicht Stylesheets.

Beispiel:

```
let button = document.querySelector("button");
button.style.backgroundColor = "blue";
button.style.padding = "10px";
```

Besonderheiten:

- CSS-Schreibweise wie `background-color` wird zu camelCase: `backgroundColor`
- Werte sind Strings inkl. Einheit: z. B. `"10px"`, `"block"`, `"#ff0000"`

---

4. Mehrere Stile auf einmal setzen: `.style.cssText`

```
box.style.cssText = "background: red; padding: 20px; border-radius: 10px;";
```

- Setzt alle Styles gleichzeitig
- Kann bestehende Inline-Stile überschreiben

---

5. Unterschied: inline-, internal-, external CSS

CSS-Typ	Anwendung	Priorität
Inline	Direkt im Element via `style`	**Höchste Priorität**
Internal	Im `<style>`-Tag im HTML-Dokument	Mittel
External	In separaten .css-Dateien	Niedrig

> **Hinweis**: Inline-Styles sollten **sparsam** eingesetzt werden – besser ist die Steuerung via CSS-Klassen.

---

6. Beispielanwendung: Sichtbarkeit toggeln

```
let modal = document.querySelector(".modal");
modal.classList.toggle("hidden");
```

Und im CSS:

```
.hidden {
  display: none;
}
```

Kapitel 7: DOM-Elemente erstellen, einfügen, klonen und entfernen

1. Elemente erstellen mit `document.createElement()`

Diese Methode erzeugt ein neues DOM-Element – es ist noch **nicht** im Dokument sichtbar, bis es eingefügt wird.

```
let newDiv = document.createElement("div");
newDiv.textContent = "Ich bin neu!";
```

Du kannst diesem neuen Element wie gewohnt Attribute, Klassen, Styles etc. zuweisen:

```
newDiv.classList.add("box");
newDiv.id = "neuesElement";
```

---

2. Elemente einfügen

`.appendChild()` – klassisch und weit verbreitet

```
document.body.appendChild(newDiv);
```

- Fügt das Element **am Ende** des Elternknotens ein
- Nur ein einzelnes Element möglich

`.append()` / `.prepend()` – moderner und flexibler

```
el.append("Text", newElement);
el.prepend(newElement);
```

- Beide Methoden erlauben mehrere Inhalte gleichzeitig (Text und/oder Elemente)
- Strings werden automatisch in Textknoten umgewandelt
- Praktisch z. B. für dynamische Listen oder Container

> Beispiel: `el.append("Hinweis: ", spanEl)` ergibt einen Text plus ein HTML-Element nebeneinander im DOM.

`.insertBefore()` – gezielt an bestimmte Stelle

```
parent.insertBefore(newElement, referenzElement);
```

- Füge `newElement` **vor** dem `referenzElement` ein

---

3. Elemente klonen mit `cloneNode()`

```
let original = document.querySelector(".card");
let kopie = original.cloneNode(true); // true = mit Kindknoten
```

- `true`: tiefes Kopieren (inkl. aller Kinder)
- `false`: nur das Element selbst

> Hinweis: Event-Listener werden beim Klonen **nicht** übernommen.

---

4. Elemente entfernen

`.removeChild()` – klassisch

```
parent.removeChild(element);
```

- Benötigt Referenz auf das Eltern-Element

`.remove()` – modern und elegant

```
element.remove();
```

- Entfernt sich selbst aus dem DOM

---

5. Beispiel: Dynamisch einfügen und löschen

HTML:

```
<div id="container"></div>
<button id="add">Hinzufügen</button>
```

JavaScript:

```
let container = document.getElementById("container");
let btn = document.getElementById("add");

btn.addEventListener("click", () => {
  let box = document.createElement("div");
  box.className = "box";
  box.textContent = "Ich bin neu";

  box.addEventListener("click", () => box.remove()); // Klick = entfernen

  container.appendChild(box);
});
```

Kapitel 8: Events und Event-Handling im DOM

Was sind Events?

Ein **Event** ist ein Ereignis, das im Browser ausgelöst wird – z. B. ein Mausklick, ein Tastendruck, das Laden einer Seite oder das Verlassen eines Eingabefeldes. JavaScript erlaubt es uns, auf solche Ereignisse **zu reagieren**, indem wir sogenannte **Event Listener** registrieren.

---

1. `addEventListener()` – Standardmethode zum Event-Handling

```
element.addEventListener("eventname", callback[, options]);
```

Argument	Bedeutung
`eventname`	Der Typ des Events, z. B. `click`, `submit`, `input`, `keydown`, ...
`callback`	Die Funktion, die ausgeführt wird, wenn das Event eintritt
`options` *(optional)*	Objekt oder Boolean, z. B. `{ once: true }` für einmalige Ausführung

> **Hinweis**: Das `eventname`-Argument ist ein **String**, der einen gültigen Event-Typ beschreibt. Eine Liste gängiger Typen findest du weiter unten in Abschnitt 3.

Beispiel:

```
let button = document.querySelector("button");
button.addEventListener("click", function(event) {
  alert("Button wurde geklickt!");
```

```
  console.log(event); // Zugriff auf das Event-Objekt
});
```

---

2. Alternativ: Event-Handler-Property direkt am Element

Du kannst auch eine Funktion direkt einer Property wie `onclick` zuweisen:

```
button.onclick = function(event) {
  alert("Klick über onclick!");
  console.log(event); // Event-Objekt auch hier verfügbar
};
```

Vergleich:

Methode	Vorteile	Nachteile
`.addEventListener()`	Mehrfach möglich, flexible	Etwas längere Syntax
`.onclick` etc.	Kürzer, einfach	Überschreibt vorherige Listener

> **Wichtig**: Auch bei der Zuweisung via `.onclick` erhält die Callback-Funktion automatisch das Event-Objekt als ersten Parameter – genau wie bei `addEventListener()`.

Gängige Event-Handler-Properties (Direktzugriff)

Property	Entspricht Event-Typ	Beschreibung
`onclick`	`click`	Mausklick
`ondblclick`	`dblclick`	Doppelklick
`onmousedown`	`mousedown`	Maustaste gedrückt
`onmouseup`	`mouseup`	Maustaste losgelassen
`onmousemove`	`mousemove`	Maus bewegt sich
`onmouseover`	`mouseover`	Maus über Element
`onmouseout`	`mouseout`	Maus verlässt Element
`onkeydown`	`keydown`	Taste gedrückt
`onkeyup`	`keyup`	Taste losgelassen
`oninput`	`input`	Eingabe verändert sich
`onchange`	`change`	Inhalt verändert (nach Fokusverlust)
`onfocus`	`focus`	Element erhält Fokus
`onblur`	`blur`	Element verliert Fokus
`onsubmit`	`submit`	Formular wird abgesendet
`onreset`	`reset`	Formular wird zurückgesetzt
`onload`	`load`	Seite oder Objekt ist geladen
`onerror`	`error`	Fehler beim Laden

> Tipp: Du kannst diese Properties direkt auf jedes DOM-Element anwenden, sofern der Event-Typ zu dem Element passt.

---

3. Gängige Event-Typen (für `addEventListener()`)

Diese **Strings** werden als erstes Argument in `addEventListener("eventname", ...)` verwendet:

Event-Typ	Beschreibung
`click`	Maus wird geklickt
`dblclick`	Doppelklick auf ein Element
`mousedown`	Maustaste wird gedrückt
`mouseup`	Maustaste wird losgelassen
`mousemove`	Maus bewegt sich über das Element
`input`	Inhalt eines Eingabefelds ändert sich
`change`	Wert ändert sich nach Fokusverlust (z. B. Select)
`submit`	Formular wird abgeschickt
`keydown`	Taste wird gedrückt
`keyup`	Taste wird losgelassen
`focus`	Element erhält Fokus
`blur`	Fokus wird verloren
`mouseover`	Maus fährt über ein Element
`mouseout`	Maus verlässt ein Element

---

4. Das Event-Objekt nutzen – Wann und wie?

Das **Event-Objekt** ist ein spezielles Objekt, das automatisch an jede Event-Callback-Funktion übergeben wird – **egal ob du `addEventListener()` oder eine Property wie `onclick` verwendest**.

Beispiel mit `addEventListener()`:

```
element.addEventListener("click", function(event) {
  console.log(event.type);      // "click"
  console.log(event.target);    // Das angeklickte Element
});
```

Beispiel mit `.onclick`:

```
element.onclick = function(event) {
  console.log(event.type);      // "click"
};
```

Eigenschaften im Event-Objekt:

Property	Beschreibung
`event.type`	Typ des Events (z. B. "click")
`event.target`	Das Element, auf das geklickt wurde
`event.currentTarget`	Das Element, auf dem der Listener registriert wurde
`event.preventDefault()`	Verhindert Standardverhalten (z. B. Formular senden)
`event.stopPropagation()`	Stoppt Event-Weiterleitung im DOM

---

5. Event Delegation

Statt jedem Kind-Element einen Listener zu geben, nutzt man **einen Listener auf dem Eltern-Element**:

```
document.querySelector("ul").addEventListener("click", function(e) {
  if (e.target.tagName === "LI") {
    console.log("Geklickt: " + e.target.textContent);
  }
});
```

- Spart Performance bei vielen Elementen
- Funktioniert auch mit dynamisch erzeugten Inhalten

---

6. Events nur einmal ausführen mit `{ once: true }`

```
button.addEventListener("click", () => console.log("nur einmal"), { once: true });
```

- Sehr praktisch z. B. für Intro-Hinweise, Popups etc.

---

7. Unterschied: `e.target` vs. `e.currentTarget`

```
el.addEventListener("click", function(e) {
  console.log("target: ", e.target);        // Das tatsächlich angeklickte Kind-Element
  console.log("currentTarget: ", e.currentTarget); // Das Element mit dem Listener
});
```

> Besonders wichtig bei **Event Delegation**, wenn `e.target` nicht gleich `e.currentTarget` ist.

Kapitel 8 – Exkurs: Das Schlüsselwort `this` in Event-Handlern

Das JavaScript-Schlüsselwort `this` spielt bei DOM-Events eine wichtige Rolle, besonders wenn du mit mehreren Elementen arbeitest und dynamisch wissen willst, **auf welches konkrete Element geklickt wurde** – z. B. bei einer Liste von `li`-Elementen.

Verhalten von `this` im DOM-Event-Handler

Klassische Funktion:

```
let items = document.querySelectorAll("li");
items.forEach(function(item) {
  item.addEventListener("click", function() {
    console.log(this); // verweist auf das angeklickte <li>
    this.classList.toggle("selected");
  });
});
```

`this` verweist auf das DOM-Element, auf dem der Event ausgelöst wurde.

Arrow Function:

```
items.forEach(item => {
  item.addEventListener("click", () => {
    console.log(this); // NICHT das <li>, sondern z. B. window
```

```
  });
});
```

In Arrow Functions ist `this` **lexikalisch gebunden** – nicht empfehlenswert für DOM-Events.

Unterschied zu `event.target`

Ausdruck	Bedeutung
`this`	Das Element, an das der Event-Handler gebunden ist
`event.target`	Das Element, das tatschächlich geklickt wurde

Beispiel mit Verschachtelung:

```
<li><strong>Wichtig</strong> Text</li>
```

```
list.addEventListener("click", function(e) {
  console.log("target:", e.target); // <strong>
  console.log("this:", this);       // <ul>
  console.log("currentTarget:", e.currentTarget); // <ul>
});
```

Fazit

- Nutze **`this` in klassischen Funktionen**, um auf das eigene DOM-Element zuzugreifen
- Vermeide `this` in **Arrow Functions** bei Event-Handling
- Verwende **`event.target`**, wenn du das Element ermitteln willst, das **konkret** betroffen war (z. B. bei Event Delegation)

Damit kannst du zielsicher und effizient Event-Handler bei mehreren DOM-Elementen einsetzen – besonders nützlich für Navigationen, ToDo-Listen und dynamische UI-Komponenten.

Kapitel 9: DOM Traversal & gezielte Navigation

DOM Traversal bezeichnet das **Durchqueren des DOM-Baums**, also das gezielte Navigieren zwischen Eltern-, Kind- und Geschwister-Elementen. Das ist besonders nützlich, wenn man von einem bekannten Ausgangselement aus andere verwandte Elemente im DOM ansprechen möchte.

---

1. Wiederholung: Navigations-Eigenschaften im DOM

Eigenschaft	Beschreibung
`parentNode`	Liefert das Elternelement eines Knotens
`childNodes`	Liefert eine NodeList aller Kindknoten (inkl. Textknoten)
`children`	Liefert eine HTMLCollection aller **Element-Kinder**
`firstChild` / `lastChild`	Erstes/letztes Kind – auch Textknoten möglich
`firstElementChild` / `lastElementChild`	Nur erste/letzte **Element-Knoten**
`previousSibling` / `nextSibling`	Vorheriges/nächstes Geschwister-**Node**
`previousElementSibling` / `nextElementSibling`	Vorheriges/nächstes **Element**

Beispiel:

```
let item = document.querySelector("li");
console.log(item.parentNode); // z.B. <ul>
console.log(item.nextElementSibling); // nächstes <li>
```

---

2. `.closest(selector)` – aufwärts im DOM suchen

Sucht das **nächstgelegene übergeordnete Element**, das auf den CSS-Selektor passt.

```
let btn = document.querySelector("button");
let card = btn.closest(".card");
```

- Praktisch für Event-Delegation oder wenn ein Element **in einem Container** steckt

---

3. `.matches(selector)` – Prüfen, ob Element einem Selektor entspricht

```
let el = document.querySelector("li");
if (el.matches(".active")) {
  console.log("Ist aktiv!");
}
```

- Liefert `true`, wenn das aktuelle Element den Selektor erfüllt
- Ideal für Filter- oder Highlight-Funktionen

---

4. Traversal in Kombination mit Events

```
document.addEventListener("click", function(e) {
  if (e.target.matches(".remove-btn")) {
    let card = e.target.closest(".card");
    card.remove();
  }
});
```

- Selektives Entfernen durch Event-Delegation + Traversal

---

5. Tipps für DOM Traversal in der Praxis

- **`children` statt `childNodes`** verwenden, wenn du nur HTML-Elemente brauchst
- Nutze `.closest()` für saubere Selektion nach oben
- Nutze `.matches()` für Bedingungen innerhalb von Event-Handlern
- Achte auf **Whitespace** und Textknoten bei `firstChild`/`childNodes`

Kapitel 11: Formulare und Validierung im DOM

Formulare sind ein zentraler Bestandteil interaktiver Webseiten. JavaScript ermöglicht es, **Formulardaten zu lesen, zu verändern und deren Gültigkeit zu prüfen** – sowohl mit eigenen Regeln als auch durch die native HTML5-Validierung.

---

1. Zugriff auf Formulardaten

Um auf die Eingabewerte in Formularen zuzugreifen, verwendet man `.value` bei den Formularelementen:

```
```

```html
<form id="login">
  <input type="text" name="username">
  <input type="password" name="password">
  <button type="submit">Login</button>
</form>
```

```js
let form = document.getElementById("login");
let username = form.elements.username.value;
```

- Jedes Eingabefeld mit `name="..."` wird automatisch über `form.elements` zugänglich
- Alternativ: Selektieren über `querySelector()` + `.value`

---

2. `submit`-Event abfangen mit `preventDefault()`

Standardmäßig lädt ein Formular die Seite neu. Um das zu verhindern, kann man `event.preventDefault()` aufrufen:

```js
form.addEventListener("submit", function(event) {
  event.preventDefault();
  console.log("Gesendet!", form.elements.username.value);
});
```

- Wichtig für Single-Page-Apps und moderne Formverarbeitung im Frontend

---

3. Native HTML5-Validierung mit `.checkValidity()`

Jedes Formularfeld kennt den eingebauten Validierungsstatus:

```js
let input = document.querySelector("input");
if (input.checkValidity()) {
  console.log("Gültige Eingabe");
} else {
  console.log("Ungültige Eingabe");
}
```

- `.checkValidity()` prüft alle HTML-Constraints (`required`, `minlength`, `pattern`, etc.)
- `.reportValidity()` zeigt zusätzlich die Standard-Fehlermeldung im Browser an

---

4. Eigene Validierungsregeln mit `.setCustomValidity()`

```js
let input = document.querySelector("input");
input.addEventListener("input", function() {
  if (input.value.length < 5) {
    input.setCustomValidity("Mindestens 5 Zeichen erforderlich");
  } else {
    input.setCustomValidity(""); // Gültig, keine Meldung
  }
});
```

- Kombination aus eigener Logik und HTML5-Validierung möglich
- Mit `.reportValidity()` kann man die Meldung sofort anzeigen lassen

---

5. Gängige Validierungsattribute (HTML)

Attribut	Beschreibung
`required`	Feld darf nicht leer sein
`minlength`	Mindestanzahl Zeichen
`maxlength`	Höchstanzahl Zeichen
`pattern`	Regulärer Ausdruck zur Eingabekontrolle
`type=email`	Prüft auf E-Mail-Format
`type=number`, `min`, `max`	Zahlenbereich definieren

---

6. Praktisches Beispiel: Login-Formular validieren & Daten absenden

```
form.addEventListener("submit", async function(e) {
  e.preventDefault();

  let user = form.elements.username.value.trim();
  let pass = form.elements.password.value;

  if (user === "") {
    form.elements.username.setCustomValidity("Bitte Benutzernamen eingeben.");
    form.elements.username.reportValidity();
    return;
  } else {
    form.elements.username.setCustomValidity("");
  }

  if (pass.length < 6) {
    form.elements.password.setCustomValidity("Mindestens 6 Zeichen erforderlich.");
    form.elements.password.reportValidity();
    return;
  } else {
    form.elements.password.setCustomValidity("");
  }

  // Datenstruktur für den Versand vorbereiten
  const formData = {
    username: user,
    password: pass
  };

  try {
    const response = await fetch("/api/login", {
      method: "POST",
      headers: {
        "Content-Type": "application/json"
      },
      body: JSON.stringify(formData)
    });

    if (!response.ok) throw new Error("Fehler beim Senden");

    const result = await response.json();
    console.log("Login erfolgreich:", result);
  } catch (err) {
    console.error("Login fehlgeschlagen:", err);
  }
});
```

Kapitel 12: Mini-Projekt – Dynamische ToDo-Liste

ToDo-Liste mit folgenden Funktionen:

- Neue Aufgaben hinzufügen
- Aufgaben als erledigt markieren
- Aufgaben löschen
- Zählung der offenen Aufgaben

Wir setzen dabei alle bisherigen DOM-Kenntnisse in die Praxis um.

Schritt 1: HTML-Struktur vorbereiten

```
<h1>ToDo-Liste</h1>
<form id="todo-form">
  <input type="text" id="todo-input" placeholder="Neue Aufgabe" required>
  <button type="submit">Hinzufügen</button>
</form>

<ul id="todo-list"></ul>
<p id="todo-count">0 Aufgaben offen</p>
```

Schritt 2: JavaScript-Grundstruktur einrichten

```
const form = document.getElementById("todo-form");
const input = document.getElementById("todo-input");
const list = document.getElementById("todo-list");
const countDisplay = document.getElementById("todo-count");

form.addEventListener("submit", function(e) {
  e.preventDefault();
  addTask(input.value);
  input.value = "";
});

function updateCount() {
  const total = list.querySelectorAll("li:not(.done)").length;
  countDisplay.textContent = `${total} Aufgaben offen`;
}
```

Schritt 3: Neue Aufgabe hinzufügen

```
function addTask(text) {
  const li = document.createElement("li");
  li.textContent = text;

  const doneBtn = document.createElement("button");
  doneBtn.textContent = "✔";
  doneBtn.className = "done-btn";
  doneBtn.title = "Als erledigt markieren";

  const deleteBtn = document.createElement("button");
  deleteBtn.textContent = "✖";
  deleteBtn.className = "delete-btn";
  deleteBtn.title = "Löschen";
```

```
  li.append(doneBtn, deleteBtn);
  list.appendChild(li);

  updateCount();
}
```

Schritt 4: Aufgabenstatus verwalten (erledigt/löschen)

```
list.addEventListener("click", function(e) {
  const li = e.target.closest("li");

  if (e.target.classList.contains("done-btn")) {
    li.classList.toggle("done");
  }

  if (e.target.classList.contains("delete-btn")) {
    li.remove();
  }

  updateCount();
});
```

Schritt 5: Einfaches CSS (optional)

```
<style>
  li.done { text-decoration: line-through; color: gray; }
  button { margin-left: 5px; }
</style>
```

Erweiterungsideen

- Aufgaben im `localStorage` speichern und beim Laden wiederherstellen
- Datum/Uhrzeit bei jeder Aufgabe anzeigen
- Editieren von Aufgaben hinzufügen
- Nach Status filtern (Alle, Offen, Erledigt)
