

Javascript Grundkurs

1. Einführung

Die Entstehungsgeschichte von JavaScript

JavaScript wurde **1995** von **Brendan Eich** bei **Netscape Communications** entwickelt. Damals war das Ziel, Webseiten interaktiver zu machen — z. B. durch einfache Überprüfungen in Formularen, ohne den Server zu belasten.

Kurze Zeitleiste

Jahr	Ereignis	Bedeutung
1995	JavaScript entsteht bei Netscape (ursprünglich „Mocha“, dann „LiveScript“)	Erste Implementierung im Browser „Netscape Navigator“
1996	Microsoft veröffentlicht „JScript“ im Internet Explorer	Beginn der Browser-Inkompatibilitäten
1997	ECMA (European Computer Manufacturers Association) standardisiert die Sprache als ECMAScript	Erste offizielle Sprachspezifikation (ES1)
2009	ECMAScript 5 veröffentlicht	Moderne Sprachfeatures wie <code>Array.forEach()</code> , <code>Object.create()</code>
2015	ECMAScript 6 (ES2015) erscheint	Der große Wendepunkt: Klassen, Module, Arrow Functions, <code>let/const</code>
Heute	Jährliche Updates (ES2016, ... ES202x)	Kontinuierliche Weiterentwicklung der Sprache

Standardisierung: W3C vs. ECMA

JavaScript ist eine **Skriptsprache**, deren **Standard** bei der Organisation **ECMA International** gepflegt wird.

- **ECMA** → Verantwortlich für den Sprachstandard **ECMAScript (ES)**
- **W3C** → Verantwortlich für **Webstandards**, z. B. HTML und CSS

Das bedeutet:

- **ECMA** legt fest, *wie die Sprache funktioniert*
- **W3C** legt fest, *wie der Browser Webseiten darstellt und wie JS darauf zugreifen darf*

Technologische Einordnung von JavaScript

JavaScript ist:

- **eine interpretierte Sprache** (kein vorheriges Kompilieren nötig),
- **dynamisch typisiert** (Datentypen werden zur Laufzeit bestimmt),
- **multi-paradigmatisch** (funktional, objektorientiert, imperativ),
- **asynchron fähig** (durch Events, Promises, `async/await`).

JavaScript ist keine „abgespeckte Version von Java“

Der Name entstand aus Marketinggründen.

Beide Sprachen haben:

- eine ähnliche Syntax,
- aber völlig verschiedene Laufzeitmodelle und Philosophien.



1.4 – Ausführung von JavaScript im Browser

Jeder Browser hat eine eigene **JavaScript-Engine**, die den Code interpretiert und ausführt:

Browser	Engine
Chrome / Edge	V8
Firefox	SpiderMonkey
Safari	JavaScriptCore („Nitro“)

Die Engine führt den Code in einem **Single Thread** aus (Ereignisschleife/Event Loop).

Alternative Ausführungsumgebungen

Neben dem Browser kann JavaScript auch **serverseitig oder systemnah** ausgeführt werden.

Umgebung	Beschreibung	Beispielhafte Nutzung
Node.js	JavaScript-Laufzeit auf Basis von V8	Serverseitige Webanwendungen, Tools, APIs
Deno	Moderne Alternative zu Node.js von Brendan Eich	Sichere Ausführung, ES Modules by default
Bun	Neue, sehr schnelle Runtime	Build-System + Laufzeitumgebung
Embedded JS (IoT)	JavaScript auf Geräten (z. B. Espruino, Tessel)	Sensorsteuerung, Microcontroller

Zusammenfassung

- JavaScript ist heute eine der wichtigsten Sprachen der Webentwicklung.
- Ursprünglich für Browser entwickelt, läuft es inzwischen überall.
- Die Sprache wird von **ECMA International** standardisiert.
- Der Browser bietet zusätzliche APIs (DOM, Events, etc.) über W3C-Spezifikationen.
- JavaScript ist flexibel, dynamisch und vielseitig — die Basis moderner Webanwendungen.

IDE – Integrierte Entwicklungsumgebung

Eine **integrierte Entwicklungsumgebung (IDE)** ist ein **Softwareprogramm**, das Entwickler*innen beim **Schreiben, Testen und Debuggen von Code** unterstützt.

Sie **bündelt mehrere Werkzeuge** in einer einzigen Anwendung, z. B.:

- **Editor** zum Schreiben des Codes
- **Compiler oder Interpreter** zum Übersetzen und Ausführen des Programms
- **Debugger** zum Finden und Beheben von Fehlern
- **Projektverwaltung** und **Versionskontrolle**

Kurz gesagt:

Eine IDE ist eine **Arbeitsumgebung**, die alle wichtigen Funktionen zum **Entwickeln von Software an einem Ort** vereint – zum Beispiel **Visual Studio Code, Eclipse** oder **PyCharm**.

2. Sprachmerkmale & Syntax

JavaScript ist **dynamisch typisiert** und **interpretiert**.

Das bedeutet:

- Variablen müssen **nicht vorab deklariert** werden, welcher Datentyp es ist (wie in C oder Java).
- Der Typ einer Variable kann sich **zur Laufzeit** ändern.

```
let value = 42;           // Zahl (number)
value = "Hallo";          // Jetzt ein String
console.log(value);       // Ausgabe: "Hallo"
```



Datentypen

JavaScript kennt **7 primitive Datentypen** und **1 Referenztyp (Object)**.

Primitive Typen

Typ	Beispiel	Beschreibung
number	42, 3.14	Alle Zahlen (ganz & Gleitkomma)
string	"Hallo" oder 'Welt'	Zeichenketten
boolean	true, false	Wahrheitswerte
undefined	let x;	Variable existiert, aber kein Wert
null	let y = null;	bewusste Leere
symbol	Symbol('id')	eindeutige, unveränderliche Werte
bigint	123n	Sehr große Ganzzahlen

Referenztyp / Non Primitive

Typ	Beispiel	Beschreibung
object	{ name: "Lisa" }, [1, 2, 3], new Date()	Sammlungen von Werten und Methoden

Operatoren

Arithmetische Operatoren

```
let sum = 5 + 3;    // 8
let rest = 10 % 3;  // 1 (Modulo)
let power = 2 ** 3; // 8 (Potenzierung)
```

Vergleichsoperatoren

```
console.log(5 == "5"); // true (Typ wird umgewandelt)
console.log(5 === "5"); // false (striker Vergleich)
```



Logische Operatoren

```
let isAdult = true;
let hasTicket = false;

console.log(isAdult && hasTicket); // false
console.log(isAdult || hasTicket); // true
```

Zuweisungsoperatoren

```
javascript

let x = 10;
x += 5;    // entspricht x = x + 5;
x *= 2;    // entspricht x = x * 2;
```

Variablen, Literale und Ausdrücke

In JavaScript werden Variablen mit `let`, `const` oder `var` deklariert.

Schlüsselwort	Bereich (Scope)	Veränderbar	Typische Nutzung
<code>var</code>	Funktionsweit	✓ Ja	veraltet – vermeiden!
<code>let</code>	Blockweit	✓ Ja	Standard für veränderliche Werte
<code>const</code>	Blockweit	✗ Nein	Konstanten & unveränderliche Referenzen

Beispiel:

```
let age = 30;
const name = "Alex";

age = 31;    // erlaubt
// name = "Ben"; ✗ Fehler: const kann nicht geändert werden
```

Literale

Ein Literal ist ein fester Wert im Code

```
"Hallo Welt"    // String-Literal
42              // Zahlenliteral
true           // Boolean-Literal
[1, 2, 3]       // Array-Literal
{ key: "value" } // Objekt-Literal
```

Ausdrücke und Anweisungen

Ein **Ausdruck** ist ein Stück Code, das **einen Wert ergibt**.

Eine **Anweisung (Statement)** ist eine **vollständige Anweisung**, die ausgeführt wird.

```
// Ausdruck:
5 + 10

// Anweisung:
let result = 5 + 10;
```

Zusammenfassung Sprachmerkmale & Syntax

- JavaScript hat **primitive** und **Referenztypen**.
- Variablen werden mit `let` und `const` deklariert.
- Operatoren steuern Logik, Vergleiche und Berechnungen.
- Typüberprüfungen erfolgen mit `typeof`.
- `===` ist immer dem `==` vorzuziehen.

Objekte vergleichen in JavaScript

Das Thema ist so wichtig, weil in JavaScript **Objekte nicht nach Inhalt, sondern nach Referenz** verglichen werden.

Lass uns das Schritt für Schritt aufdröseln.

Beispiel:

```
let a = { name: "Alex" };  
let b = { name: "Alex" };  
  
console.log(a === b); // ❌ false
```

Warum?

Weil JavaScript prüft, ob **beide Variablen auf dasselbe Objekt im Speicher zeigen** — und nicht, ob sie denselben **Inhalt** haben.

3. Funktionen & Scopes

Was ist eine Funktion?

Eine **Funktion** ist ein wiederverwendbarer Codeblock, der eine Aufgabe ausführt.

Man kann ihr Daten übergeben (**Parameter**) und sie kann ein Ergebnis zurückgeben (**Return-Wert**).

```
function greet() {  
  console.log("Hallo Welt!");  
}  
  
greet(); // Funktionsaufruf
```




Funktionen mit Parametern und Rückgabewerten

Funktionen können **Parameter** entgegennehmen:

```
function greetUser(name) {  
  console.log("Hallo, " + name + "!");  
}  
  
greetUser("Lisa");    // Ausgabe: Hallo, Lisa!  
greetUser("Alex");    // Ausgabe: Hallo, Alex!
```

Eine Funktion kann aber auch einen Wert zurückgeben

```
function add(a, b) {  
  return a + b;  
}  
  
let result = add(5, 3);  
console.log(result); // 8
```

Verschiedene Arten, Funktionen zu schreiben

Funktionendeklaration

```
function sayHello() {  
  console.log("Hallo!");  
}
```

Funktionsausdruck / Name Funktionen

```
const sayHello = function() {  
  console.log("Hallo!");  
};
```



Unterschied zwischen Deklarationen und Ausdruck

Deklarationen werden beim Start des Skripts „gehoisted“ (hochgezogen) — sie sind **überall im Code verfügbar**.
Ausdrücke dagegen **nicht**.

Arrow Funktionen (ES6)

```
const add = (a, b) => a + b;  
console.log(add(4, 6)); // 10
```

Advanced Funktionen Sektion

In JavaScript gibt es mehrere Arten von Funktionen, die je nach Anwendungsfall eingesetzt werden können. Hier sind die häufigsten Funktionsarten:

1. Normale (deklarierte) Funktionen

```
function greet(name) {  
  return `Hallo, ${name}!`;  
}
```

- Werden mit dem Schlüsselwort `function` deklariert.
- Haben einen eigenen Namen und können vor ihrer Definition aufgerufen werden (Hoisting).

2. Anonyme Funktionen

```
const greet = function(name) {  
  return `Hallo, ${name}!`;  
};
```



- Haben keinen Namen.
- Werden häufig in Variablen gespeichert oder als Argumente übergeben.

3. Arrow-Funktionen (Pfeilfunktionen)

```
const greet = (name) => `Hallo, ${name}!`;
```

- Kürzere Syntax für Funktionen.
- Vererben this vom umgebenden Kontext (kein eigenes this).
- Ideal für Callback- und Kurzfunktionen

4. Methoden

```
const person = {  
  greet(name) {  
    return `Hallo, ${name}!`;  
  }  
};
```

- Funktionen, die als Methoden eines Objekts definiert werden.
- Werden direkt auf dem Objekt aufgerufen.

5. Konstruktor-Funktionen

```
function Person(name) {  
  this.name = name;  
  this.greet = function() {  
    return `Hallo, ${this.name}!`;  
  };  
}
```

```
const person = new Person('Max');
```

- Werden mit new aufgerufen.
- Dienen der Erstellung von Objekten.

6. Generator-Funktionen

```
function* numberGenerator() {  
  let i = 0;  
  while (true) {  
    yield i++;  
  }  
}
```

```
const gen = numberGenerator();  
console.log(gen.next().value); // 0  
console.log(gen.next().value); // 1
```



- Verwenden das function*-Schlüsselwort.
- Geben mit yield Werte nacheinander zurück

7. Asynchrone Funktionen

```
async function fetchData() {  
  const response = await fetch('https://api.example.com');  
  return await response.json();  
}
```

- Verwenden das async-Schlüsselwort.
- Arbeiten mit await für asynchrone Operationen.

8. Callback-Funktionen

```
function doTask(callback) {  
  console.log("Aufgabe wird ausgeführt...");  
  callback();  
}
```

```
doTask(() => console.log("Callback wird aufgerufen!"));
```

- Werden als Argumente an andere Funktionen übergeben.
- Häufig verwendet bei asynchronen oder ereignisbasierten Aufgaben.

9. Selbstaufrufende Funktionen (IIFE - Immediately Invoked Function Expressions)

```
(function() {  
  console.log("Diese Funktion ruft sich sofort auf!");  
})();
```

- Direkt nach ihrer Definition aufgerufen.
- Häufig verwendet, um einen eigenen Scope zu schaffen.

4. Scope: Gültigkeitsbereich von Variablen

Welche „Scopes“ gibt es:

- globaler Scope



- lokaler Scope
- block Scope

Globaler Scope

Außerhalb einer Funktion oder eines Blocks deklarierte Variablen sind **überall sichtbar**.

```
let message = "Hallo aus dem globalen Scope!";

function printMessage() {
  console.log(message); // Zugriff erlaubt
}

printMessage();
```

Lokaler (Funktions-)Scope

Innerhalb einer Funktion deklarierte Variablen sind **nur dort sichtbar**.

```
function showNumber() {
  let number = 42;
  console.log(number);
}

showNumber();
// console.log(number); ❌ Fehler: number ist hier nicht definiert
```

Block-Scope (let und const)

let und const gelten **nur innerhalb eines Blocks** (z. B. innerhalb von {}).

```
if (true) {
  let inside = "Ich bin im Block!";
  console.log(inside); // funktioniert
}
// console.log(inside); ❌ funktioniert nicht
```

⚠️ **var** ignoriert Blockgrenzen – daher heute **nicht mehr empfohlen**.

Typumwandlung

JavaScript wandelt Werte automatisch zwischen Typen um (**implizite Konvertierung**). Man kann sie aber auch bewusst durchführen (**explizite Konvertierung**).

Implizit

javascript

```
console.log("5" + 3); // "53" → String-Verkettung
console.log("5" - 3); // 2   → "5" wird zu Zahl
```

Explizit

javascript

```
let num = Number("42");
let str = String(42);
let bool = Boolean(0); // false
```

Zusammenfassung

- **Funktionen** sind wiederverwendbare Codeblöcke mit Parametern und Rückgabewerten.
- Es gibt **Funktionsdeklarationen**, **Ausdrücke** und **Arrow Functions**.
- **Scope** definiert, wo Variablen sichtbar sind.
- **let/const** → blockweise sichtbar, **var** → funktionsweit.
- **Typumwandlungen** können automatisch oder explizit erfolgen.

5. Kontrollstrukturen & Schleifen

Verzweigungen & bedingte Anweisungen

Programme müssen oft Entscheidungen treffen.

Dazu verwendet man **bedingte Anweisungen**, die auf **Bedingungen (true/false)** reagieren.

IF / ELSE

```
let temperature = 25;

if (temperature > 30) {
  console.log("Es ist sehr heiß!");
} else if (temperature > 20) {
  console.log("Angenehm warm.");
} else {
  console.log("Kühl heute!");
}
```

Hinweis:

Bedingungen können mit logischen Operatoren (&&, |, !) kombiniert werden.

```
let isRaining = true;
let hasUmbrella = false;

if (isRaining && !hasUmbrella) {
  console.log("Du wirst nass!");
}
```

switch – Alternative zu vielen if-Abfragen

```
let day = "Montag";

switch (day) {
  case "Montag":
    console.log("Wochenstart!");
    break;
  case "Freitag":
    console.log("Fast Wochenende!");
    break;
  default:
    console.log("Ein normaler Tag.");
}
```

Break; ist wichtig – sonst werden die folgenden Fälle ebenfalls ausgeführt.

Schleifen

Schleifen wiederholen Codeblöcke, solange eine Bedingung wahr ist.

for-Schleife

Typisch, wenn man **weiß**, wie oft der Code wiederholt werden soll.

```
for (let i = 0; i < 5; i++) {  
  console.log("Durchlauf Nummer:", i);  
}
```

while-Schleife

Läuft, **solange** eine Bedingung erfüllt ist.

```
let count = 0;  
  
while (count < 3) {  
  console.log("Zähler:", count);  
  count++;  
}
```

do...while

Wird **mindestens einmal ausgeführt**, auch wenn die Bedingung zu Beginn falsch ist.

```
let number = 5;  
  
do {  
  console.log("Zahl ist:", number);  
  number++;  
} while (number < 3);
```

Iteration durch Arrays und Objekte

for...of – Elemente in einem Array durchlaufen

```
let fruits = ["Apfel", "Banane", "Kirsche"];  
  
for (let fruit of fruits) {  
  console.log("Frucht:", fruit);  
}
```

for...in – Eigenschaften eines Objekts durchlaufen

```
let person = { name: "Alex", age: 30, city: "Berlin" };

for (let key in person) {
  console.log(key, "→", person[key]);
}
```

Zusammenfassung

- **if / else** und **switch** treffen Entscheidungen im Code.
- **for**, **while**, **do...while** wiederholen Anweisungen.
- **for...of** und **for...in** sind nützlich für Arrays und Objekte.
- Schleifen sind der Grundbaustein für Datenverarbeitung und Logik.

SPREAD Operator

Der **Spread-Operator** in JavaScript (dargestellt durch drei Punkte `...`) ist eine sehr nützliche Syntax, die es erlaubt, **Elemente eines Arrays oder Eigenschaften eines Objekts zu "entpacken"**, um sie an eine andere Stelle zu kopieren, kombinieren oder weiterzuverarbeiten.

Durch die `...` Notierung wird eine „shallow Copy“ der Datei erstellt. Diese kann dann mit neuen Parametern ergänzt werden.

Spread auf Objekten

```
const user = {username: „Peter“}
```

um das user object mit dem age – key zu erweitern schreibt man folgendes:

```
const user2 = {...user, age: 45}
```

somit ist user2 dann {username: „Peter“, age: 45}

Die JavaScript Standard API





JavaScript bringt viele **eingebaute Objekte und Methoden** mit, die **ohne externe Bibliotheken** verfügbar sind.

Diese Sammlung heißt **Standard API** oder auch **Standard Library**.

Zu den wichtigsten gehören:

- String
- Number
- Math
- Date
- Array
- Object
- JSON

In diesem Modul konzentrieren wir uns auf:

 Strings,  Number,  Math,  Date

Strings

Strings sind Zeichenketten (Text) und werden mit " . . ." oder ' . . .' geschrieben.

Nützliche String-Methoden

Methode	Beschreibung	Beispiel
<code>.length</code>	Länge des Strings	<code>"Hallo".length</code> → 5
<code>.toUpperCase()</code>	Alles in Großbuchstaben	<code>"hi".toUpperCase()</code> → "HI"
<code>.toLowerCase()</code>	Alles in Kleinbuchstaben	<code>"Hi".toLowerCase()</code> → "hi"
<code>.includes()</code>	Prüft, ob Teilstring vorkommt	<code>"Hallo Welt".includes("Welt")</code> → true
<code>.indexOf()</code>	Gibt die Position zurück	<code>"Hallo".indexOf("o")</code> → 4
<code>.substring(a,b)</code>	Schneidet Teilstring heraus	<code>"JavaScript".substring(0,4)</code> → "Java"
<code>.split()</code>	Teilt String in Array	<code>"a,b,c".split(",")</code> → ["a", "b", "c"]
<code>.trim()</code>	Entfernt Leerzeichen vorne und hinten	<code>" hallo ".trim()</code> → "hallo"

Arbeiten mit Numbers

Zahlen sind in JavaScript vom Typ `number` (egal ob ganze Zahl oder Kommazahl).

Nützliche Number-Methoden & Funktionen

Methode/Funktion	Beschreibung	Beispiel
<code>Number("42")</code>	String → Zahl	<code>"42"</code> → 42
<code>parseInt("10.5")</code>	Ganzzahl auslesen	10
<code>parseFloat("10.5")</code>	Kommazahl auslesen	10.5
<code>.toFixed(n)</code>	Rundet und gibt String zurück	<code>(1.2345).toFixed(2)</code> → "1.23"
<code>isNaN(value)</code>	Prüft, ob kein gültiger Zahlwert	<code>isNaN("abc")</code> → true

Das Math-Objekt

Das eingebaute `Math`-Objekt stellt **mathematische Konstanten und Funktionen** bereit.

```

console.log(Math.PI);           // 3.141592653589793
console.log(Math.sqrt(16));     // 4
console.log(Math.pow(2, 3));    // 8
console.log(Math.abs(-10));     // 10
console.log(Math.round(4.7));   // 5
console.log(Math.floor(4.7));   // 4
console.log(Math.ceil(4.1));    // 5

```

Arbeiten mit Datum & Zeit

JavaScript bietet das Objekt `Date`, um mit Datum und Uhrzeit zu arbeiten.

Aktuelles Datum & Uhrzeit

```
let now = new Date();
console.log(now); // z. B. 2025-10-06T09:15:00.000Z
console.log(now.toString()); // "Mon Oct 06 2025"
console.log(now.toLocaleString()); // "06.10.2025, 11:15:00"
```

Einzelne DATUM WERTE auslesen

```
let today = new Date();
console.log(today.getFullYear()); // Jahr
console.log(today.getMonth()); // Monat (0-11!)
console.log(today.getDate()); // Tag
console.log(today.getHours()); // Stunde
```

Eigenes Datum erstellen



```
let birthday = new Date(1995, 4, 15); // 15. Mai 1995
console.log(birthday.toString());
```

Zusammenfassung

- Die **Standard API** bietet mächtige Werkzeuge für Text, Zahlen, Datum und Mathematik.
- **String-Methoden** helfen bei der Textverarbeitung.
- **Math** liefert Funktionen und Zufallswerte.
- **Date** ermöglicht Datumsberechnung und Zeitangaben.
- Diese Objekte sind **grundlegend für fast jedes Programm** – auch in TypeScript.

JavaScript im Browser

Das Document Object Model (DOM)

Das **DOM** ist eine **hierarchische Baumstruktur**, die den Aufbau einer HTML-Seite beschreibt. Jedes HTML-Element wird als **Objekt** im DOM dargestellt, das JavaScript ansprechen kann.

```
<body>
  <h1 id="title">Willkommen!</h1>
  <p class="info">Dies ist ein Beispieltext.</p>
</body>
```

Zugriff auf DOM-Elemente

Methode	Beschreibung	Beispiel
document.getElementById()	Zugriff über ID	document.getElementById("title")
document.querySelector()	Zugriff über CSS-Selektor	document.querySelector(".info")
document.querySelectorAll()	Zugriff auf mehrere	document.querySelectorAll("p")



Methode

Beschreibung Elemente

Beispiel

Merke:

- `textContent` → reiner Text (sicherer, kein HTML).
- `innerHTML` → auch HTML-Inhalt möglich (vorsichtig bei Benutzereingaben).

Beispiel DOM Manipulation

```
<h2 id="status">Status: offline</h2>
<button id="btn">Verbinden</button>

<script>
  const statusText = document.getElementById("status");
  const button = document.getElementById("btn");

  button.addEventListener("click", function() {
    statusText.textContent = "Status: online";
    statusText.style.color = "green";
  });
</script>
```

Events und Event-Handler

Ein **Event** ist eine Aktion, die im Browser

passiert — z. B.:

Ereignis	Beschreibung
click	Benutzer klickt auf ein Element
input	Benutzer tippt etwas in ein Eingabefeld
submit	Formular wird abgeschickt
mouseover	Maus bewegt sich über ein Element
keydown	Taste wird gedrückt

Events in JavaScript

In JavaScript sind Events Aktionen oder Vorkommnisse, die während der Interaktion mit einer Webseite auftreten, wie das Klicken auf einen Button, das Bewegen der Maus, das Laden einer Seite oder das Drücken einer Taste. Sie ermöglichen die Interaktivität von Webseiten.

Wichtige Eigenschaften:

1. **Event Listener:** Ein Mechanismus, der auf ein bestimmtes Ereignis wartet und darauf reagiert.
 - Beispiel: `addEventListener("click", function)` fügt einem Element einen Listener für Klicks hinzu.
2. **Event Typen:**
 - Mausereignisse: `click`, `dblclick`, `mouseover`, `mouseout`, `mousemove`
 - Tastaturereignisse: `keydown`, `keyup`, `keypress`
 - Formularereignisse: `submit`, `focus`, `blur`, `change`
 - Fensterereignisse: `load`, `resize`, `scroll`, `unload`

Beispiel:

```
const button = document.querySelector('button');
button.addEventListener('click', (event) => {
    alert('Button wurde geklickt!');
    console.log(event); // Details zum Ereignis
});
```

Dynamische Inhalte & DOM-Erweiterung

Mit JavaScript lassen sich auch **neue Elemente erstellen oder entfernen**.

```
let list = document.querySelector("#todoList");
let newItem = document.createElement("li");
newItem.textContent = "Neuer Punkt";
list.appendChild(newItem);
```

Wichtig:

`createElement()` erzeugt ein neues Element im Speicher.

`appendChild()` fügt es in die DOM-Struktur ein.

Best Practices: Unobtrusive JavaScript & Barrierefreiheit

Trennung von Struktur, Stil und Logik

Bereich	Technologie	Beispiel
Struktur	HTML	<code><h1>, <form></code>
Stil	CSS	<code>.button { color: blue; }</code>
Verhalten	JavaScript	<code>addEventListener("click", ...)</code>

→ Das macht Code **übersichtlicher**, **wartbarer** und **barriereärmer**.

Barrierefreiheit (Accessibility)

- Interaktive Elemente (Buttons, Links) **immer semantisch korrekt** wählen
- **ARIA-Attribute** einsetzen, wenn nötig
- Fokus-Management beachten (`element.focus()`)
- Fehlermeldungen klar und deutlich formulieren

Zusammenfassung

- Das **DOM** ist die Brücke zwischen HTML und JavaScript.
- Mit `querySelector()` & Co. greift man auf Elemente zu.
- **Events** lösen Reaktionen im Code aus.
- **Formulare** können mit JavaScript überprüft werden.
- **Unobtrusive JavaScript** trennt Logik von Struktur → Best Practice.



Ausblick: Nächste Schritte in deiner Lernreise

a) TypeScript – Der nächste logische Schritt

TypeScript ist eine Erweiterung von JavaScript mit **statischer Typisierung** und **modernen Entwicklungswerkzeugen**.

Es hilft, Fehler früh zu erkennen und Projekte skalierbarer zu machen.

Empfohlene Themen:

- TypeScript-Grundlagen (Typen, Interfaces, Klassen)
- Typinferenz und Generics
- TypeScript im Browser oder in Node.js-Projekten
- Konfiguration mit `tsconfig.json`

💡 **Vorteil:** Alles, was du über JavaScript gelernt hast, gilt weiterhin — TypeScript erweitert es nur.

b) Moderne Frameworks & Libraries

Wenn du mit TypeScript oder Vanilla JavaScript sicher bist, kannst du Frameworks lernen, um komplexe Webanwendungen zu entwickeln:

Framework / Library	Beschreibung	Einsatzbereich
React	Komponentensystem von Meta	Interaktive Web-Apps
Vue.js	Einsteigerfreundlich & flexibel	Single-Page-Apps
Angular	Vollständiges Framework von Google	Große Enterprise-Projekte
Next.js / Nuxt.js	Server-Side Rendering & Routing	Moderne Fullstack-Entwicklung

c) Node.js & Backend-Entwicklung

Mit **Node.js** läuft JavaScript auch auf dem Server.



Lerne:

- eigene Webserver mit **Express.js**
- REST-APIs entwickeln
- Datenbanken (MongoDB, PostgreSQL) anbinden
- Asynchronität und Error Handling verstehen

Damit wirst du zum **Fullstack Developer**.

d) Tools & Workflow

Professionelle Entwickler:innen nutzen Tools, um Code effizient und standardisiert zu schreiben:

Tool	Nutzen
VS Code	Standard-Editor für JS/TS
npm / yarn / pnpm	Paketmanagement
ESLint & Prettier	Codequalität & Formatierung
Git & GitHub	Versionskontrolle & Zusammenarbeit
Vite / Webpack	Build & Bundling