Einführung in die JavaScript-Programmierung

Fokus: Frontend-Entwicklung

1. Einführung, Geschichte und Status Quo

JavaScript (JS) wurde ursprünglich 1995 von Brendan Eich bei Netscape in nur 10 Tagen entwickelt. Was als einfache Skriptsprache begann, um Webseiten etwas Dynamik zu verleihen, hat sich zur wichtigsten Sprache des Webs entwickelt.

Wichtig ist die Abgrenzung: **Java ist nicht JavaScript**. (Ein populärer Vergleich: Java verhält sich zu JavaScript wie „Car“ zu „Carpet“).

Der Standard: ECMAScript

JavaScript ist eine Implementierung des **ECMAScript**-Standards. Wenn man heute von „modernem JavaScript“ spricht, meint man meist **ES6** (ECMAScript 2015) und neuer. Diese Versionen brachten Klassen, Module, Arrow-Functions und `let`/`const`, was die Sprache für große Softwareprojekte tauglich machte.

JavaScript Engine

JS ist eine interpretierte (bzw. Just-in-Time kompilierte) Sprache. Im Browser führt eine Engine den Code aus. Die bekannteste ist **V8** (Google Chrome), die auch die Basis für Node.js (Backend) und Deno bildet.

2. Variablen und Scopes (Geltungsbereiche)

In älterem JS-Code findet man häufig `var`. In modernem Code gilt die Regel: **Vermeiden Sie `var` vollständig.**

2.1. let, const und var

- **`var`**: Hat einen **Function Scope**. Variablen sind innerhalb der gesamten Funktion sichtbar, auch bevor sie deklariert wurden (Hoisting), was zu schwer findbaren Fehlern führt.
- **`let`**: Hat einen **Block Scope** (wie in C# oder Java). Die Variable existiert nur innerhalb des geschweiften Klammernpaars `{ ... }`. Sie ist veränderbar.
- **`const`**: Wie `let` (Block Scope), aber die Zuweisung ist einmalig.

2.2. Mutable vs. Immutable bei const

Wichtig: `const` schützt die *Bindung* der Variable, nicht den *Inhalt* (bei Referenztypen).

```
const pi = 3.1415;
pi = 3; // Fehler: Assignment to constant variable.

const student = { name: "Max" };
student.name = "Moritz"; // ERLAUBT! Das Objekt wird mutiert.
// student = { name: "Moritz" }; // FEHLER: Neue Zuweisung verboten.
```

Best Practice: Nutzen Sie standardmäßig `const`. Nutzen Sie `let` nur, wenn Sie wissen, dass sich der Wert ändern muss (z.B. Schleifenzähler).

3. Datentypen und Operatoren

JavaScript ist **dynamisch typisiert**. Variablen haben keinen festen Typ, Werte schon.

3.1. Primitives vs. Objects

- **Primitives:** `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`. (Call-by-Value).
- **Objects:** Arrays, Funktionen, Klassen, Objekte. (Call-by-Reference).

3.2. Der Vergleichs-Wahnsinn (Equality)

JS versucht bei Vergleichen oft, Typen automatisch umzuwandeln (Type Coercion).

- **== (Loose Equality):** Wandelt Typen um. **Gefährlich.**
  - `0 == "0"` -> `true`
  - `false == "0"` -> `true`
  - `[] == 0` -> `true`
- **=== (Strict Equality):** Vergleicht Typ UND Wert. **Immer verwenden.**
  - `0 === "0"` -> `false`

Fail-Fast Regel: Verwenden Sie in Ihrem Code ausschließlich `===` und `!==`. Es gibt kaum Gründe für `==`.

3.3. Truthy und Falsy

In `if`-Abfragen werden Werte in Boolean umgewandelt.

- **Falsy:** `false`, `0`, `""` (leerer String), `null`, `undefined`, `NaN`.
- **Truthy:** Alles andere (auch leere Arrays `[]` und leere Objekte `{}`!).

4. Funktionen

Funktionen sind „First-Class Citizens“. Sie können wie Variablen behandelt, übergeben und zurückgegeben werden.

4.1. Funktions-Definitionen

```
// Klassisch (Function Declaration)
function addiere(a, b) {
    return a + b;
}

// Als Variable (Function Expression)
const subtrahiere = function(a, b) {
    return a - b;
};
```

4.2. Argumente

JS prüft die Anzahl der Argumente nicht. Fehlende Argumente sind `undefined`. Modern löst man Default-Werte direkt in der Signatur:

```
function begruesse(name = "Gast") {
    console.log(`Hallo, ${name}`); // Template String (wie C# Interpolation)
}
```

4.3. Arrow Functions (Lambda)

Seit ES6 gibt es die kürzere Arrow-Syntax. Sie ist analog zu C# Lambdas `() => {}`.

```
const malZwei = (x) => {
    return x * 2;
};

// Kurzform (Implicit Return bei einer Expression)
const malDrei = x => x * 3;
```

Wichtiger Unterschied: Arrow Functions haben kein eigenes `this`. Sie erben `this` aus dem umgebenden Kontext (lexical scoping). Das macht sie ideal für Event-Handler innerhalb von Klassen.

5. Arrays und Higher-Order Functions

Arrays in JS sind dynamisch (wie `List<T>` in C#) und können gemischte Datentypen enthalten (sollte man aber vermeiden).

Wichtig: Statt `for`-Schleifen nutzen wir in modernem JS meist funktionale Methoden.

Gegeben sei:

```
const zahlen = [1, 2, 3, 4, 5];
```

5.1. Wichtige Methoden

- **`forEach`**: Iteriert über Elemente (Rückgabe: void).

  ```
  zahlen.forEach(z => console.log(z));
  ```

- **`map`**: Transformiert jedes Element und erstellt ein **neues** Array (Projektion, wie C# `Select`).

  ```
  const quadrate = zahlen.map(z => z * z); // [1, 4, 9, 16, 25]
  ```

- **`filter`**: Erstellt ein **neues** Array mit Elementen, die die Bedingung erfüllen (Prädikat, wie C# `Where`).

  ```
  const gerade = zahlen.filter(z => z % 2 === 0); // [2, 4]
  ```

- **`find`**: Gibt das **erste** Element zurück, das passt (oder `undefined`).

  ```
  const treffer = zahlen.find(z => z > 3); // 4
  ```

6. Das DOM (Document Object Model)

Wenn wir Frontend programmieren, programmieren wir fast immer das **DOM**: also die HTML-Struktur, die der Browser als Baum im Speicher hält.

6.1. Was ist das DOM?

- Das DOM ist eine **Baumstruktur** aus Nodes (Elemente, Text, Kommentare, ...).
- `document` ist der Einstiegspunkt.
- Jedes HTML-Element wird zu einem DOM-Element-Objekt (z.B. ein `<button>` wird zu `HTMLButtonElement`).

Wichtig: Das DOM ist **nicht** die HTML-Datei, sondern die *Live*-Repräsentation im Browser.

6.2. Wann darf ich auf das DOM zugreifen?

Wenn dein Skript vor dem HTML geladen wird, sind Elemente noch nicht vorhanden. Sicher ist:

```
document.addEventListener('DOMContentLoaded', () => {
    // Hier existiert das HTML im DOM
});
```

Alternativ (häufig in Übungen): Script-Tag ganz am Ende von `<body>`.

6.3. DOM-Elemente finden (Selectors)

Modern arbeiten wir fast immer mit CSS-Selektoren:

```
const button = document.querySelector('.buy-button');
const items = document.querySelectorAll('.cart-item');
```

- `querySelector(...)` gibt **das erste** passende Element (oder `null`).
- `querySelectorAll(...)` gibt eine `NodeList` (iterierbar).

Fail-Fast Regel: Rechne mit `null`, wenn ein Element nicht existiert.

```
const modal = document.querySelector('#modal');
if (!modal) throw new Error('Modal fehlt im HTML');
```

6.4. Inhalte ändern (Text vs. HTML)

```
titleEl.textContent = 'Hallo!';
```

- `textContent`: sicher (keine HTML-Interpretation).
- `innerHTML`: bequem, aber potenziell **unsicher** (XSS) und fehleranfällig.

Regel: Nutze `textContent` und DOM-Erzeugung (`createElement`) statt `innerHTML`, außer du kontrollierst den Inhalt zu 100%.

6.5. Elemente erzeugen/entfernen

```
const li = document.createElement('li');
li.textContent = 'Neuer Eintrag';
listEl.appendChild(li);

// Entfernen
li.remove();
```

Für "neu rendern" ist oft besser als `innerHTML = ''`:

```
listEl.replaceChildren(); // leert den Container
```

6.6. Events (User-Interaktion)

```
button.addEventListener('click', (event) => {
    console.log('geklickt', event.currentTarget);
});
```

- `event.target`: das *tatsächlich* geklickte Element (z.B. ein `<span>` im Button).
- `event.currentTarget`: das Element, an dem der Listener hängt.

Event Delegation (sehr wichtig): Ein Listener auf dem Container statt 100 Listener auf Kinder.

```
listEl.addEventListener('click', (e) => {
    const button = e.target.closest('button[data-action="remove"]');
    if (!button) return;

    const id = button.dataset.id;
    console.log('remove', id);
});
```

6.7. Klassen-Manipulation zur UI-Steuerung

Die beste UI-Architektur im Vanilla-Frontend ist oft:

- JavaScript setzt/entfernt **CSS-Klassen** (State)
- CSS steuert Aussehen/Animation/Responsiveness

Die zentrale API ist `element.classList`:

```
panel.classList.add('is-open');
panel.classList.remove('is-open');
panel.classList.toggle('is-open');
panel.classList.toggle('is-open', true);  // explizit setzen
panel.classList.contains('is-open');
```

Beispiel A: Active Navigation HTML:

```
<nav class="nav">
    <a class="nav-link is-active" href="#home">Home</a>
    <a class="nav-link" href="#about">About</a>
</nav>
```

JS:

```
const links = document.querySelectorAll('.nav-link');

links.forEach(link => {
    link.addEventListener('click', (e) => {
        e.preventDefault();
        links.forEach(l => l.classList.remove('is-active'));
        link.classList.add('is-active');
    });
});
```

CSS (UI-Logik in CSS):

```
.nav-link.is-active {
    background-color: green;
    color: white;
}
```

Beispiel B: Modal / Overlay öffnen und schließen JS:

```
const openBtn = document.querySelector('[data-open-modal]');
const closeBtn = document.querySelector('[data-close-modal]');
const modal = document.querySelector('[data-modal]');

if (!openBtn || !closeBtn || !modal) throw new Error('Modal-Markup unvollstaendig');

const openModal = () => {
    modal.classList.add('is-open');
    document.body.classList.add('no-scroll');
};

const closeModal = () => {
    modal.classList.remove('is-open');
    document.body.classList.remove('no-scroll');
};

openBtn.addEventListener('click', openModal);
closeBtn.addEventListener('click', closeModal);
modal.addEventListener('click', (e) => {
    if (e.target === modal) closeModal(); // Klick auf Backdrop
});
```

CSS:

```
.modal {
    display: none;
}

.modal.is-open {
    display: block;
}

body.no-scroll {
    overflow: hidden;
}
```

Beispiel C: Loading/Disabled State bei Buttons

```
const saveBtn = document.querySelector('#save');
if (!saveBtn) throw new Error('save button fehlt');

const setLoading = (isLoading) => {
    saveBtn.classList.toggle('is-loading', isLoading);
    saveBtn.disabled = isLoading;
```

```
};

saveBtn.addEventListener('click', async () => {
    try {
        setLoading(true);
        // await fetch(...) / await irgendwas
    } finally {
        setLoading(false);
    }
});
```

6.8. Mini-Regeln fuer gutes DOM-UI

- Halte DOM-Refs zentral (z.B. `const elements = {...}`) statt überall neu zu suchen.
- Schreibe so, dass UI-State *sichtbar* ist: `is-open`, `is-active`, `is-loading`, `has-error`.
- Manipuliere das Design nicht per JS (kein `el.style...` als Default). Nutze Klassen.
- Fehler frueh abfangen: fehlende Nodes, falsche Selektoren, ungueltige Inputs.
- Denke an Barrierefreiheit: wenn du etwas ausblendest, dann sinnvoll (z.B. `hidden`, `aria-expanded`).

7. Objektorientierung: Klassen

JS basierte ursprünglich auf Prototypen. ES6 führte das `class` Keyword ein, das „Syntactic Sugar“ ist, sich aber fast wie C# anfühlt.

```
class Auto {
    // Private Felder beginnen mit # (echte Runtime Privacy!)
    #seriennummer;

    constructor(marke, baujahr) {
        this.marke = marke;   // Public Property
        this.baujahr = baujahr;
        this.#seriennummer = crypto.randomUUID();
    }

    // Getter
    get beschreibung() {
        return `${this.marke} (Bj. ${this.baujahr})`;
    }

    starten() {
        console.log(`Auto ${this.#seriennummer} startet...`);
    }
}

const meinAuto = new Auto("VW", 2025);
// meinAuto.#seriennummer -> Syntax Error (Privat)
```

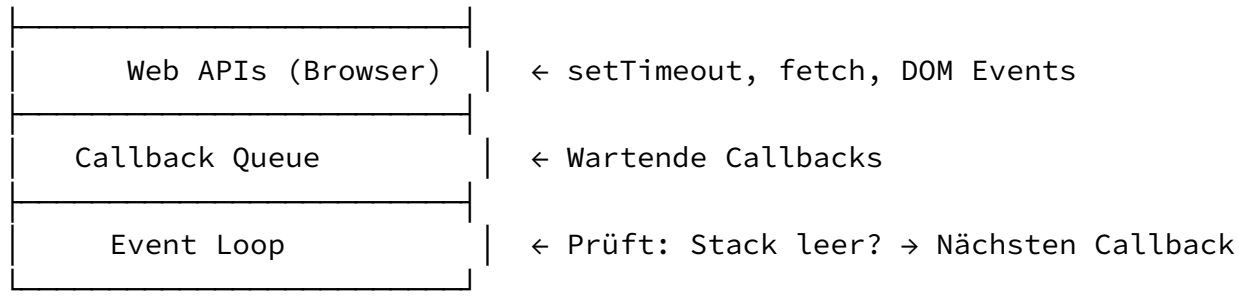
8. Asynchrone Programmierung

JavaScript läuft in einem einzigen Thread (Single Threaded Event Loop). Blockierende Operationen (wie Netzwerkanfragen) würden das UI einfrieren. Asynchrone Programmierung ist deshalb essentiell für moderne Webanwendungen.

8.1. Der Event Loop verstehen

Der Event Loop ist JavaScripts Herzstück. Er ermöglicht das “scheinbare” Multitrotasking in einem Single-Thread:

```
┌─────────────────────────┐
│        Call Stack       │  ← Wo Code ausgeführt wird
```



Wichtig: `async` Code läuft nicht parallel (keine Threads), sondern ” später”.

8.2. Von Callbacks zu Promises

Historisch: Callbacks (Callback Hell)

```
// Alt und unübersichtlich
ladeDaten(function(data) {
    verarbeiteDaten(data, function(result) {
        speichereResult(result, function() {
            console.log("Fertig!");
        });
    });
});
```

Modern: Promises ermöglichen flache, lesbare Ketten.

8.3. Promises im Detail

Ein `Promise` repräsentiert einen Wert, der jetzt, später oder nie verfügbar ist. Zustände: `pending` (laufend) `fulfilled` (Erfolg) oder `rejected` (Fehler).

```
function ladeDaten() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const erfolg = true;
            if (erfolg) {
                resolve("Daten geladen");
            } else {
                reject(new Error("Laden fehlgeschlagen"));
            }
        }, 1000);
    });
}
```

Promise Methoden Promise Chaining mit `.then()`:

```
ladeDaten()
    .then(daten => {
        console.log("Schritt 1:", daten);
        return daten.toUpperCase();
    })
    .then(daten => {
        console.log("Schritt 2:", daten);
        return verarbeiteWeiter(daten);
    })
    .catch(error => {
        console.error("Fehler in der Kette:", error);
    })
    .finally(() => {
        console.log("Cleanup (immer ausgeführt)");
    });
```

Promise Kombinationen:

```
const promise1 = fetch('/api/users');
const promise2 = fetch('/api/posts');
const promise3 = fetch('/api/comments');

// Promise.all - Wartet auf ALLE (fail-fast)
Promise.all([promise1, promise2, promise3])
    .then(([users, posts, comments]) => {
        console.log("Alles geladen");
    })
    .catch(error => {
        console.log("Einer ist fehlgeschlagen");
    });

// Promise.allSettled - Wartet auf alle (egal ob Erfolg/Fehler)
Promise.allSettled([promise1, promise2, promise3])
    .then(results => {
        results.forEach(result => {
            if (result.status === 'fulfilled') {
                console.log("Erfolg:", result.value);
            } else {
                console.log("Fehler:", result.reason);
            }
        });
    });

// Promise.race - Erstes Ergebnis (Timeout-Pattern)
const timeout = new Promise((_, reject) =>
    setTimeout(() => reject(new Error('Timeout')), 5000)
);

Promise.race([ladeDaten(), timeout])
    .then(daten => console.log("Rechtzeitig geladen"))
    .catch(err => console.log("Zu langsam oder Fehler"));
```

8.4. Async / Await Syntax

`async/await` ist "Syntactic Sugar" für Promises - es sieht aus wie synchroner Code, ist aber nicht-blockierend.

```
async function appStart() {
    try {
        console.log("Lade...");
        const daten = await ladeDaten();
        console.log("Ergebnis:", daten);
    } catch (error) {
        console.error("Es gab ein Problem:", error);
    } finally {
        console.log("Cleanup");
    }
}
```

Wichtige Regeln: * `async` vor einer Funktion macht sie automatisch zu einer Promise-returning Funktion * `await` kann nur in `async` Funktionen verwendet werden (oder in Top-Level-Modules) * Ein `throw` in einer `async` Funktion rejectet das Promise

Parallel vs. Seriell

```
// ❌ Seriell (langsam): Eine nach der anderen
async function langsam() {
    const user = await fetchUser();
```

```
    const posts = await fetchPosts();   // Wartet auf user!
    const comments = await fetchComments(); // Wartet auf posts!
    return { user, posts, comments };
}

// ✅ Parallel (schnell): Alle gleichzeitig
async function schnell() {
    const [user, posts, comments] = await Promise.all([
        fetchUser(),
        fetchPosts(),
        fetchComments()
    ]);
    return { user, posts, comments };
}
```

8.5. Fetch API - Praktisches Beispiel

```
async function getUserData(userId) {
    const url = `https://api.example.com/users/${userId}`;

    try {
        const response = await fetch(url);

        // HTTP-Fehler prüfen (404, 500, etc.)
        if (!response.ok) {
            throw new Error(`HTTP ${response.status}: ${response.statusText}`);
        }

        const data = await response.json();
        return data;

    } catch (error) {
        console.error("API-Fehler:", error);
        // Entscheiden: Weiterwerfen oder Default-Wert?
        throw error;
    }
}

// POST-Request mit JSON
async function createUser(userData) {
    const response = await fetch('/api/users', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify(userData)
    });

    if (!response.ok) throw new Error('Erstellung fehlgeschlagen');
    return await response.json();
}
```

8.6. Fehlerbehandlung Patterns

Pattern 1: Try-Catch mit spezifischen Fehlertypen

```
async function robusteFunktion() {
    try {
        const daten = await ladeDaten();
        return daten;
    } catch (error) {
```

```
        if (error.name === 'NetworkError') {
            console.log('Netzwerkproblem - Retry?');
        } else if (error.name === 'TimeoutError') {
            console.log('Timeout - später nochmal versuchen');
        } else {
            console.log('Unbekannter Fehler:', error);
        }
        throw error; // Oder: return defaultWert;
    }
}
```

Pattern 2: Result-Objekte statt Exceptions

```
async function ladeDatenSafe() {
    try {
        const daten = await fetchData();
        return { success: true, data: daten };
    } catch (error) {
        return { success: false, error: error.message };
    }
}

// Verwendung
const result = await ladeDatenSafe();
if (result.success) {
    console.log(result.data);
} else {
    console.log("Fehler:", result.error);
}
```

Pattern 3: Retry mit Exponential Backoff

```
async function fetchWithRetry(url, maxRetries = 3) {
    for (let i = 0; i < maxRetries; i++) {
        try {
            return await fetch(url);
        } catch (error) {
            if (i === maxRetries - 1) throw error;
            const delay = Math.pow(2, i) * 1000; // 1s, 2s, 4s
            console.log(`Retry ${i + 1} in ${delay}ms...`);
            await new Promise(resolve => setTimeout(resolve, delay));
        }
    }
}
```

8.7. Häufige Fehler vermeiden

```
// ❌ Vergessen zu await
async function falsch() {
    const daten = fetchData(); // Promise, nicht das Ergebnis!
    console.log(daten); // Promise {<pending>}
}

// ✅ Richtig mit await
async function richtig() {
    const daten = await fetchData();
    console.log(daten); // Die tatsächlichen Daten
}

// ❌ Catch ohne Rethrow (stille Fehler)
async function problematisch() {
    try {
        return await riskanteOperation();
```

```
    } catch (e) {
        console.log("Fehler ignoriert"); // Daten werden nie zurückgegeben!
    }
}

// ✅ Richtig: Fehler weitergeben oder default
async function besser() {
    try {
        return await riskanteOperation();
    } catch (e) {
        console.log("Fehler:", e);
        return { error: true, message: e.message }; // Oder: throw e;
    }
}

// ❌ Array.map mit async (erzeugt Array von Promises)
async function mapFalsch(userIds) {
    return userIds.map(async id => await fetchUser(id));
    // Rückgabe: [Promise, Promise, Promise] - nicht die Daten!
}

// ✅ Richtig: Promise.all warten
async function mapRichtig(userIds) {
    const promises = userIds.map(id => fetchUser(id));
    return await Promise.all(promises);
    // Rückgabe: [User1, User2, User3]
}
```

8.8. Microtasks und Reihenfolge

```
console.log('1');

setTimeout(() => console.log('2'), 0);

Promise.resolve().then(() => console.log('3'));

console.log('4');

// Ausgabe: 1, 4, 3, 2
// Warum? Promises (Microtasks) haben Priorität vor setTimeout (Macrotasks)
```

9. Architektur moderner Frontend-Anwendungen

Um ohne Frameworks (wie React oder Angular) sauberen, wartbaren Code zu schreiben, strukturieren wir unsere Applikation strikt nach dem Prinzip **State-Driven UI**. Wir trennen Daten (State) von der Darstellung (DOM).

Hier ist der **7-Punkte-Plan** für eine Vanilla-JS Applikation:

1. APPLICATION STATE OBJECT (Single Source of Truth)

Der gesamte Zustand der App liegt in einem zentralen Objekt. Nicht im HTML verstreut.

```
const state = {
    todos: [],
    filter: 'all', // 'all', 'active', 'completed'
    loading: false
};
```

2. DOM Node Refs

Wir speichern Referenzen auf wichtige statische HTML-Elemente, um nicht ständig `querySelector` aufzurufen (Performance & Übersicht).

```
const elements = {
    input: document.querySelector('#todo-input'),
    listContainer: document.querySelector('#todo-list'),
    addButton: document.querySelector('#add-btn'),
};
```

3. DOM Node Creation Fn's (Helper)

Statt HTML-Strings manuell zusammenzubauen (Sicherheitsrisiko XSS), nutzen wir kleine Helper-Funktionen, die DOM-Elemente erzeugen.

```
const createEl = (tag, className, text = '') => {
    const el = document.createElement(tag);
    if (className) el.className = className;
    el.textContent = text;
    return el;
};
```

4. RENDER FN (State -> UI)

Eine Funktion `render()`, die aufgerufen wird, wann immer sich Daten ändern. Sie leert den Container und baut ihn basierend auf dem aktuellen `state` neu auf. Das garantiert Synchronität.

```
const render = () => {
    elements.listContainer.innerHTML = ''; // Reset

    state.todos.forEach(todo => {
        const li = createEl('li', 'todo-item', todo.text);
        // Visuelle Logik
        if (todo.done) li.classList.add('completed');
        elements.listContainer.appendChild(li);
    });
};
```

5. EVENT HANDLERS (Business Logic)

Funktionen, die auf User-Input reagieren, den `state` aktualisieren und danach (!) `render()` aufrufen.

```
const handleAdd = () => {
    const text = elements.input.value;
    if (!text) return; // Fail-Fast

    state.todos.push({ text, done: false });
    elements.input.value = ''; // Form Reset
    render(); // UI Update
};
```

6. INIT BINDINGS (Verdrahtung)

Hier werden die Event-Listener einmalig beim Start gesetzt.

```
const bindEvents = () => {
    elements.addButton.addEventListener('click', handleAdd);
    // Optional: Enter-Taste im Input
    elements.input.addEventListener('keyup', (e) => {
        if (e.key === 'Enter') handleAdd();
    });
};
```

7. INITIAL RENDER

Der Einstiegspunkt (“Main”), der alles startet.

```
const init = () => {
    console.log("App startet...");
    bindEvents();
    render(); // Initialer Status (z.B. leere Liste) anzeigen
};

// Start nach Laden des DOMs
document.addEventListener('DOMContentLoaded', init);
```

Autor: T3 Chat | Kontext: HTL Spengergasse Teaching Material