Tag 1 JavaScript: Grundlagen und erste Schritte

1. Was ist JavaScript?

JavaScript ist eine interpretierte, dynamisch typisierte Programmiersprache, die ursprünglich für die Interaktivität von Webseiten im Browser entwickelt wurde. Heute wird sie sowohl client- als auch serverseitig eingesetzt (z. B. mit Node.js oder Deno).

Eigenschaften:

- JavaScript wird **nicht kompiliert**, sondern **direkt ausgeführt** (interpretiert), meist durch den Browser.
- Es ist **dynamisch typisiert**: Variablen können jederzeit ihren Datentyp ändern.
- Die Sprache ist **eventgesteuert** – viele Aktionen passieren als Reaktion auf Benutzereingaben oder Zeitereignisse.
- JavaScript ist **standardisiert durch ECMAScript** (aktuell Version ES2023) und dadurch in allen modernen Browsern weitgehend kompatibel.

Einsatzgebiete:

- Interaktive Webseiten (DOM-Manipulation, z. B. Menüs, Slider, Modalfenster)
- Validierung von Formularen direkt im Browser
- Serverseitige Programmierung mit Node.js
- Mobile Apps mit Frameworks wie React Native
- Spieleentwicklung, Datenvisualisierung, Browserautomatisierung u. v. m.

Historischer Kontext

JavaScript wurde 1995 in nur zehn Tagen von Brendan Eich entwickelt und zuerst unter dem Namen **LiveScript** veröffentlicht. Kurz darauf wurde es zu "JavaScript" umbenannt, um vom damaligen Hype um Java zu profitieren – obwohl beide Sprachen technisch fast nichts miteinander zu tun haben. Die Weiterentwicklung erfolgt heute durch **ECMA International**, der Sprachstandard heißt **ECMAScript**.

---

2. Interpretation vs. Kompilierung

Unterschied erklärt

Interpreter	**Compiler**
Führt Code Zeile für Zeile aus	Übersetzt kompletten Code vor der Ausführung
Fehler erscheinen zur Laufzeit	Fehler vor Ausführung sichtbar
JavaScript, Python	C, C++, Rust
Flexibel, dynamisch	Effizient, stark typisiert

JavaScript ist – wie auch Python – eine **interpretierte Sprache**. Der Code wird also **nicht vorab in Maschinensprache übersetzt**, sondern direkt beim Laden im Browser ausgeführt.

Ein Vorteil: Änderungen am Code sind sofort testbar, ohne Kompilierungsschritt. Ein Nachteil: Viele Fehler zeigen sich **erst zur Laufzeit** und können zu unerwartetem Verhalten führen.

Moderne Browser-Engines (wie V8 in Chrome) verwenden heute zusätzlich sogenannte **Just-in-Time (JIT) Compiler**, um häufig genutzten Code zu optimieren.

3. JavaScript im Browser ausführen

a) Inline in HTML

JavaScript lässt sich direkt in eine HTML-Datei einbetten. Diese Variante nennt man **inline JavaScript**. Sie ist nützlich für einfache Experimente oder zum Testen einzelner Funktionen.

```
<!DOCTYPE html>
<html>
<head><title>JS Test</title></head>
<body>
  <script>
    console.log("Hallo aus JavaScript!");
  </script>
</body>
</html>
```

Best Practice:

Das `<script>`-Tag sollte **möglichst am Ende des** `<body>` eingefügt werden – direkt **vor dem schließenden** `</body>`**-Tag**.

Warum?

- JavaScript wird direkt beim Parsen ausgeführt – steht es oben, kann es den Aufbau der Seite blockieren.
- Wenn JavaScript auf ein Element zugreifen will, das im HTML noch gar nicht geladen wurde, entsteht ein Fehler.
- Am Ende des `<body>` ist sichergestellt, dass der gesamte DOM geladen ist und alle Elemente im Code verfügbar sind.

Alternativ kann man im `<head>` das Attribut `defer` verwenden:

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

Damit wartet der Browser mit der Ausführung, bis das HTML vollständig geladen ist – optimal für externe Skripte.

b) Externe Datei einbinden

Größere JavaScript-Programme sollten **nicht inline** geschrieben, sondern in **separate** `.js`**-Dateien** ausgelagert werden.

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

Dies macht den Code übersichtlicher, besser wartbar und ermöglicht Wiederverwendbarkeit.

c) Direkt in der Browser-Konsole

Jeder moderne Browser hat eine integrierte Entwicklerkonsole, in der JavaScript direkt eingegeben und ausgeführt werden kann. Diese eignet sich hervorragend zum Testen einzelner Codezeilen oder für Experimente.

Zugriff: Rechtsklick auf eine Seite → "Untersuchen" → "Konsole" (alternativ `F12` oder `Strg+Shift+I`)

```
console.log("Testausgabe in der Konsole");
```

Auch Variablen, Funktionen und sogar DOM-Elemente können hier direkt manipuliert werden.

4. Aufbau eines JavaScript-Programms

Grundstruktur & Syntax

Ein JavaScript-Programm besteht aus einer oder mehreren Anweisungen (Statements), die nacheinander ausgeführt werden. Die Syntax ist dabei an C-Sprachen angelehnt (wie C, Java, C++), unterscheidet sich aber in vielen Details.

- Jede Anweisung kann (aber muss nicht) mit einem **Semikolon** `;` abgeschlossen werden.
- Der Code ist **case-sensitive**: `Name` und `name` sind unterschiedliche Bezeichner.
- Kommentare dienen der Dokumentation und werden vom Interpreter ignoriert:

```
// Einzeiliger Kommentar
/* Mehrzeiliger
   Kommentar */
```

Beispiel: Ein einfaches Programm

```
console.log("Willkommen zu JavaScript");
document.write("Hallo Welt auf der Seite");
```

Diese beiden Zeilen geben Text in der Konsole bzw. im HTML-Dokument aus. `console.log()` ist eine wichtige Funktion zum Debuggen – sie funktioniert wie `print()` in Python.

Benutzerinteraktion

JavaScript bietet einfache Möglichkeiten, mit Benutzer:innen zu interagieren:

```
alert("Hallo Benutzer!");
let name = prompt("Wie heißt du?");
console.log("Hallo, " + name);
```

- `alert()` zeigt eine einfache Nachricht an.
- `prompt()` fordert den Benutzer auf, einen Text einzugeben – dieser wird als `String` zurückgegeben.

Diese Funktionen sind nützlich, um grundlegende Interaktion zu testen – in der Praxis ersetzt man sie später meist durch HTML-Formulare und DOM-Manipulation.

(Weitere Kapitel folgen im selben ausführlichen Stil...)

---

5. Client- vs. serverseitige Programmierung

JavaScript kann nicht nur im Browser laufen, sondern auch auf dem Server – z. B. mit Node.js oder Deno. Es ist wichtig, den Unterschied zwischen diesen beiden Einsatzorten zu kennen, denn sie haben unterschiedliche Möglichkeiten und Einschränkungen.

Clientseitiges JavaScript (im Browser)

- Läuft direkt im Browser des Benutzers
- Hat Zugriff auf:
  - HTML-Dokument (DOM)
  - Benutzerinteraktion (z. B. Klicks, Tastatur)
  - Browser-APIs (z. B. `window`, `document`, `fetch`)
- **Sicherheitsbeschränkt**: Kein Zugriff auf Festplatte oder lokale Dateien (außer Sandbox)

Beispiel:

```
alert("Dies ist clientseitiger Code");
```

Dieser Code läuft direkt im Browser und kann z. B. eine Interaktion mit der Seite starten.

Serverseitiges JavaScript (z. B. mit Node.js)

- Wird auf einem Server ausgeführt (nicht im Browser)
- Hat Zugriff auf:
  - Dateisystem
  - Netzwerk, Datenbanken
  - Backend-Logik und Routing
- Kein Zugriff auf das DOM oder Benutzeroberflächen

Beispiel:

```
// server.js
console.log("Ich laufe serverseitig");
```

Ausführung im Terminal:

```
node server.js
```

Merksatz:

> **Clientseitig** = alles, was im Browser läuft.
> **Serverseitig** = alles, was im Hintergrund auf einem Server geschieht.

Beide Seiten können miteinander kommunizieren – z. B. über **HTTP-Anfragen**. Das ist die Grundlage moderner Web-Apps.

---

6. JavaScript-Umgebungen und erste Tools

Wie kann ich JavaScript schreiben und testen?

Es gibt mehrere Möglichkeiten, JavaScript zu entwickeln – je nach Ziel, Kenntnisstand und Projektgröße.

a) Browser-Konsole (zum schnellen Testen)

- Direkt zugänglich, kein Setup nötig
- Ideal für kurze Experimente oder das Nachvollziehen einzelner Konzepte

b) Online-Editoren

- z. B. jsfiddle.net, codepen.io
- Vorteile:
  - Kein Installationsaufwand
  - Ergebnisse sofort sichtbar
  - Ideal für Prototypen, kleine Projekte oder gemeinsames Arbeiten

c) Lokale Entwicklungsumgebung mit VS Code

- Empfehlung für ernsthafte Projekte und den späteren Berufsalltag
- Kombination aus:
  - **VS Code** als Code-Editor
  - **Live Server** Extension (zur lokalen Vorschau im Browser)
  - Projektstruktur mit `index.html`, `main.js`, `style.css`

Beispiel Projektstruktur:

```
```

```
projekt-ordner/
├── index.html
├── main.js
└── style.css
```

d) JupyterLab mit Deno (nur optional)

- Funktioniert, ist aber eher umständlich
- Nicht realitätsnah für Webentwicklung
- Kein DOM-Zugriff möglich – daher für JavaScript-Einstieg nicht empfohlen

---

7. JavaScript unter der Haube – Event Loop & Engine

JavaScript läuft im Browser – aber wie funktioniert das eigentlich genau? Das Verständnis der internen Mechanik hilft später beim Debugging und beim Verständnis asynchroner Vorgänge.

JavaScript-Engines

Jeder moderne Browser hat eine eigene „Engine", die JavaScript-Code interpretiert:

- Chrome → **V8**
- Firefox → **SpiderMonkey**
- Safari → **JavaScriptCore**

Die Engine sorgt dafür, dass der Code analysiert, optimiert und ausgeführt wird. Teilweise wird er just-in-time kompiliert, also während der Ausführung in Maschinencode umgewandelt – für bessere Performance.

Was ist die Event Loop?

JavaScript ist **single-threaded**, d. h. es kann nur eine Anweisung gleichzeitig ausführen. Dennoch scheint es, als könnte es „mehrere Dinge gleichzeitig" tun – z. B. auf Benutzereingaben reagieren, während ein Timer läuft. Möglich wird das durch die **Event Loop**.

Bestandteile der Event Loop:

1. **Call Stack (Aufrufstapel)**:
    - Hier landen alle synchronen Funktionen.
    - Neue Funktionen werden oben drauf gelegt (Last-In-First-Out).
2. **Web APIs (im Browser)**:
    - Asynchrone Funktionen wie `setTimeout`, `fetch`, DOM-Events werden vom Browser übernommen und unabhängig vom Stack verarbeitet.
3. **Callback Queue (Warteschlange)**:
    - Sobald ein asynchroner Vorgang abgeschlossen ist, landet der zugehörige Callback hier.
4. **Die Event Loop selbst**:
    - Prüft ständig, ob der Call Stack leer ist.
    - Wenn ja, nimmt sie den nächsten Eintrag aus der Callback Queue und führt ihn aus.

Beispiel zur Veranschaulichung:

```
console.log("Start");
setTimeout(() => {
  console.log("Timeout beendet");
}, 0);
console.log("Ende");
```

Ausgabe:

```
Start
Ende
Timeout beendet
```

Obwohl der Timeout auf 0 ms gesetzt ist, wird er **nach** dem synchronen Code ausgeführt – weil `setTimeout` über die Web API abgewickelt wird und der Callback erst **nach dem aktuellen Stack** ausgeführt wird.

Häufiger Anfängerfehler: Annehmen, dass `setTimeout(..., 0)` sofort ausgeführt wird – tatsächlich wartet es auf den nächsten „Tick" der Event Loop.

---

8. JavaScript als Sprache – Besonderheiten & Abgrenzung zu Python

JavaScript unterscheidet sich in vielen Punkten von Python – auch wenn es auf den ersten Blick ähnlich aussieht.

Besondere Merkmale von JavaScript:

- **Alles ist ein Objekt** – sogar Funktionen und Arrays
- **Funktionen sind „first-class citizens"** – sie können in Variablen gespeichert, als Argumente übergeben und verschachtelt werden
- **Keine festen Typen** – Datentypen können sich zur Laufzeit ändern
- **Lose Fehlerbehandlung** – oft wird falscher Code trotzdem ausgeführt (z. B. `undefined + 3` ergibt `NaN`)

Abgrenzung zu Python

Aspekt	JavaScript	Python
Typisierung	Dynamisch, locker	Dynamisch, strenger
Alles ist Objekt?	Ja, inkl. Arrays, Funktionen	Nur Klasseninstanzen
Funktionen	First-Class, `=>` Syntax möglich	Auch First-Class, `def`
OOP	Prototypenbasiert	Klassenbasiert
Listen	Arrays mit eigenen Methoden	Listen mit vielen Operatoren

Beispiel: Objekt in JavaScript

```
let person = {
  name: "Maria",
  alter: 28
};
console.log(person.name); // "Maria"
```

Dies ist ein Objekt – aber **nicht** wie eine Python-Klasse, sondern eher wie ein **Dictionary mit Verhalten**. Auch Arrays oder Funktionen verhalten sich wie Objekte, was sehr viel Flexibilität bringt – aber auch Missverständnisse verursachen kann.

9. Übungen (Vertiefung & Anwendung)

Übung 1: Erstes Programm

Schreibe ein JavaScript-Programm, das "Hallo JavaScript-Welt" in der Konsole ausgibt.

Übung 2: Interaktion

Bitte den Benutzer über `prompt` um seinen Namen und begrüße ihn über `console.log`.

Übung 3: Inline vs. extern

Erstelle zwei HTML-Seiten – eine mit inline JavaScript, eine mit externer JS-Datei.

Übung 4: Kommentare

Kommentiere ein Beispielprogramm vollständig mit einzeiligen und mehrzeiligen Kommentaren.

Übung 5: Client vs. Server

Schreibe je ein Beispiel für client- und serverseitiges JavaScript und erkläre den Unterschied.

Übung 6: HTML + JS kombinieren

Erstelle eine HTML-Seite, die beim Laden per `alert` eine Nachricht anzeigt und danach ein Eingabefeld über `prompt` abfragt.

Übung 7: Fehler provozieren

Erzeuge absichtlich einen Syntaxfehler (z. B. fehlende Klammer) und beobachte die Fehlermeldung in der Konsole.

Übung 8: Projektstruktur aufbauen

Lege einen Projektordner an mit `index.html`, `main.js`, `style.css`, installiere VS Code & Live Server und starte dein erstes Mini-Projekt.

Übung 9: Interpreter vs. Compiler

Erkläre den Unterschied in eigenen Worten und nenne Beispiele für beide Varianten.

Übung 10: Event Loop verstehen

Schreibe ein Programm mit `setTimeout`, `console.log` und veranschauliche die Reihenfolge der Ausgabe.

Übung 11 (Bonus): Python vs. JS – was ist anders?

Vergleiche mit einem Partner JavaScript und Python anhand einer Beispielaufgabe – z. B. Schleife oder Funktion.