Einführung in JavaScript

Introduction into Web Development

Lorenz Glißmann

SoSe 2025

Kapitelübersicht

In diesem Kapitel geht es um folgende Themen:

1. Einstieg in JavaScript
2. JavaScript-Ökosystem
3. TypeScript
4. Browser-APIs

Einstieg in JavaScript

Überblick

Annahme

- Programmierkenntnisse sind vorhanden
- *tendenziell* Java und/oder Python

Ziel

- Bestehende Fähigkeiten auf JavaScript übertragen
  - Was ist gleich?
  - Was ist anders?
- Modernes JavaScript kennenlernen
  - Besonderheiten und Best Practices
  - Programmierparadigmen und -stil

JavaScript: Eine kurze Geschichte

- 1995: JavaScript von Brendan Eich in 10 Tagen erstellt
  - Integrierte Skriptsprache für Netscape Navigator
  - Inspiriert von Scheme (Lisp) und Java
- 1996: Internet Explorer 3.0 mit JScript veröffentlicht
- 1997: JavaScript als ECMAScript standardisiert
- 1999: ECMAScript 3
- Version 4 wurde aufgegeben
- 2009: ECMAScript 5
- 2015: ECMAScript 6 (ES6)
- seitdem: jährliche Updates

Ab 2015 wurde die Sprache grundlgend überarbeitet.

JavaScript: Anwendungsbereiche

JavaScript hat inzwischen viele Anwendungsbereiche:

Im Browser:

- Interaktive Webseiten
- Dynamische Webseiten
- Single-Page Applications
- Anwendungsentwicklung

Außerhalb des Browsers:

- Server-seitig (z.B. Node.js)
- Mobile Apps (z.B. React Native)
- Desktop-Anwendungen (z.B. Electron)
- Skripte und Automatisierung

Wir starten mit JavaScript erstmal ohne Browser / Web!

JavaScript Runtimes

- Es gibt mehrere JavaScript-Engines
  - V8 (Chromium)
  - SpiderMonkey (Firefox)
  - JavaScriptCore (Safari)
- Es gibt mehrere JavaScript-Runtimes, um JavaScript ohne Browser auszuführen
  - Node.js (verwendet V8)
  - Deno (verwendet V8)
  - Bun (verwendet JavaScriptCore)
  - Electron[1] (verwendet V8)

Um JavaScript zu lernen, verwenden wir *Node.js* und *Bun*.

[1] Zielgruppe: Desktop GUI-Applikationen wie VSCodium

JavaScript ausführen


```bash
# node aus Ubuntu Packetquellen (Stark veraltet)
sudo apt install -y nodejs npm
# besser:
curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
# und für bun:
curl -fsSL https://bun.sh/install | bash
```

Kommandozeile:

```bash
node # interaktive session (REPL)
node <filename>.js # Datei ausführen
```

```bash
bun repl # interaktive session (REPL)
bun <filename>.js # Datei ausführen
```

Im Browser:

⟹ Developer Tools (F12)
⟹ Tab Console
Achtung: das ist die Umgebung des aktuellen Tabs!

Charakterisierung von JavaScript

- Üblicherweise interpretiert, oft JIT-kompiliert
- Hat außer dem Namen nicht viel mit Java zu tun
- Hochsprachig, dynamisch typisiert
- Unterstützt viele Paradigmen:
  - Imperativ / Prozedural
  - Funktional (inspiriert von Lisp)
  - Ereignisgesteuert
  - Objektorientiert (prototypenbasiert)

Wenn wir von JavaScript sprechen, meinen wir normalerweise mindestens ECMAScript 6 (ES6).

learnxinyminutes.com/de/javascript

Was unterscheidet JavaScript von anderen Sprachen?

- Ereignisgesteuerte Ausführung (Event-driven execution model)
- Nebenläufigkeit ohne Parallelität (Single-threaded)
- Starke Unterstützung für funktionale Programmierung
  - C-ähnliche Syntax mit funktionalen Programmierelementen
- Datenmodellierung (Algebraische Datentypen)
  - Einfache Basistypen flexibel zusammensetzen
  - Data-oriented design[2]
- Prototypenbasierte Objektorientierung
  - Objekte erben von anderen Objekten
  - Keine Klassen im klassischen Sinne

[2] Fokus auf Transformation von Werten, nicht auf Typen/Klassen/Objekten

Datentypen: Primitive Typen

- Variablen: `let`, `const`, (`var`, *veraltet*)
- Booleans: `true`, `false`
- Spezielle Werte: `null`, `undefined`
- Zahlen: immer 64-Bit IEEE 754 Doubles, d.h.
  - Gleichheit immer mit Epsilon
  - `0.1 + 0.2 !== 0.3`
  - `NaN !== NaN`
- *keine einzelnen Zeichen*
- Strings: UTF-16

```js
let number = 1;
const PI = 3.14159;
let x = Infinity;
let y = -Infinity;
let notANumber = NaN;
let isTrue = true;
let name = 'Alice';
let nothing = null;
let notDefined = undefined;
```

Datentypen: Mit Werten arbeiten


```js
1 + 1; // = 2
0.1 + 0.2; // = 0.30000000000000004
10 * 2; // = 20
5 / 2; // = 2.5
(1 + 3) * 2; // = 8
5 % 2; // = 1
Math.floor(5 / 2); // = 2
'a' + "b"; // = 'ab'
true && false; // = false
true || false; // = true
!true; // = false
```

```js
1 === 1; // = true
"5" == 5; // = true
1 !== 1; // = false
1 >= 2; // = false
"a" < "b"; // = true
"ab".length; // = 2
"ab".charAt(1); // = 'b'
```

Automatische Typkonvertierung

`==` und `!=` vermeiden!

Datentypen: Komplexe Typen

Arrays:

- Listen von Werten
- Dynamisch, heterogen
- Index: 0-basiert

```js
const a = [1, 2, 3, 4, 5];
a[0]; // = 1
a[1] = 5;
a.push(6);
a; // = [1, 5, 3, 4, 5, 6]
a.length; // = 6
```

Objekte:

- Key-Value-Paare
  (key: string, value: beliebig)

```js
const person = {
  name: 'Alice',
  age: 30
};
person.name; // = 'Alice'
person['age']; // = 30
person.city = 'Berlin';
```

Array-/Objektinhalte liegen auf dem Heap (Referenztypen) ⇒ werden nicht implizit kopiert!

Objekte und Arrays decken die wichtigsten Datenstrukturen ab:

- Arrays:
  - Listen, Tupel
  - Stack (`.push()`, `.pop()`)
  - Queue (`.push()`, `.shift()`)
  - Sparse Array (mit Lücken)
- Objekte:
  - Records/Structs
  - (Hash-)Maps (aber es gibt auch `Map`)
  - Set (aber es gibt auch `Set`)
  - Objekte (im Sinne von OOP)

Obwohl es speziellere Datenstrukturen gibt, sind Arrays und Objekte oft ausreichend.

```js
'Alice' === "Alice"
"Sag mal \"Hallo\"" // Escaping
'Sag mal "Hallo"'   // Escaping
`3 x 3 =
 ${3*3}`// Multiline-Template-String-Literale (Backticks)
```

Strings sind unveränderlich (immutable).

- `s.length`
- `s.charAt(i)`
- `s.substring(i, j)`
- `s.split(separator)`
- `s.replace(search, replace)`
- `s.toLowerCase()`, `s.toUpperCase()`

Funktionen

```js
// Funktionen deklarieren
function add(a, b) {
    return a + b;
}
// Arrow-Funktionen (ES6+)
const add2 = (a, b) => {
    return a + b;
}
// Kurzform (body ist ein Ausdruck)
const add3 = (a, b) => a + b;
```

Eigenschaften:

- Funktionen sind normale Werte
- Können als Argumente übergeben werden
- Fehlende Argumente sind `undefined`
- Zusätzliche Argumente werden ignoriert
- Arrow-Funktionen unterscheiden sich leicht

Datentypen: typeof-Operator


```js
typeof 123 === 'number';
typeof 'Hallo' === 'string';
typeof true === 'boolean';
typeof undefined === 'undefined';
typeof null === 'object';
typeof {} === 'object';
typeof [] === 'object';
typeof function() {} === 'function';
let x;
typeof x === 'undefined';
x = {};
typeof x["name"] === 'undefined';
```

Uninitialisierte Variablen sind `undefined`.

```js
if (bedingung) {
    // ...
} else if (andereBedingung) {
    // ...
} else {
    // ...
}

// Ternärer Operator
avg = n === 0 ? 0 : sum/n;
```

```js
switch (wert) {
    case 1:
        // ...
        break;
    case 2:
        // ...
        break;
    default:
        // ...
}
```

Funktioniert mit Zahlen und Strings.

Ausgabe

```js
console.log('Hallo, Welt!');
console.error('Fehlermeldung');
```

`console` ist ein globales Objekt, das in Browsern und Node.js verfügbar ist.

Auf der Kommandozeile werden StdOut und StdErr verwendet.

Iterativer Kontrollfluss


```js
while (bedingung) { }
const a = [];
for (let i = 0; i < 10; i++) {
    a.push(i*i);
}
for (const element of a) {
    console.log(element);
}
for (const i in a) {
    console.log(i, a[i]);
}
a.forEach(el => console.log(el));
```

- while-Schleifen
- for/of über Iterables (Arrays, Strings, Maps, Sets)
- for/in über Index/Key von Arrays/Objekten
- forEach für Arrays

Iterativer Kontrollfluss


```js
while (bedingung) { }
const a = [];
for (let i = 0; i < 10; i++) {
    a.push(i*i);
}
for (const element of a) {
    console.log(element);
}
for (const i in a) {
    console.log(i, a[i]);
}
a.forEach(el => console.log(el));
```

- while-Schleifen
- for/of über Iterables (Arrays, Strings, Maps, Sets)
- for/in über Index/Key von Arrays/Objekten
- forEach für Arrays

In der Praxis braucht man oft keine Schleifen, sondern nutzt Array-Methoden.

Entgegen einer verbreiteten Annahme ist die Antwort nicht einfach “Rekursion”.

Entgegen einer verbreiteten Annahme ist die Antwort nicht einfach “Rekursion”.

```js
const a = [1, 3, 4, 5, 6, 7, 8, 9, 10];
const sq = a.map(x => x * x);
const ev = a.filter(x => x % 2 === 0);
const sum = a.reduce((acc, x) => acc + x, 0);
const three = a.find(x => x > 2);
// Andere: some, every, sort, reverse, slice, ...
```

Diese Methoden decken viele Anwendungsfälle für Iteration / Schleifen ab.

Keine dieser Methoden verändert das Array $\Longrightarrow$ Immutability.

Higher-Order Functions und Callbacks

Higher-Order Functions

Funktionen, die mit anderen Funktionen arbeiten (als Argumente oder Rückgabewerte).

Callbacks

Eine Funktion, die als Argument übergeben wird um später aufgerufen zu werden.

Dieser Programmierstil ist in JavaScript sehr nützlich und weit verbreitet:

Ereignisse (Event-Handler)

```
.addEventListener('click', cb)
```

Transformationen

z.B. `.map(cb)`, `.filter(cb)`, `.reduce(cb)`

Asynchrone Operationen

```
fs.readFile(file, cb)
```

Netzwerkanfragen, Dateien, Sleep, …

Ergebnisse zurück geben ohne return

- Teil-/Zwischenergebnisse,
- Fehlerbehandlung `onError`

Funktionsfabriken:

```js
function greet(greeting) {
    return (name) => greeting + " " + name;
}
```

Objekte mit privaten Variablen:

```js
function counter() {
    let count = 0;
    return {
        get: () => count,
        inc: () => count++
    };
}
```

```js
const c = counter();
c.inc();
c.inc();
console.log(c.get());  // 2
```

Arrays-Syntax und -Methoden


```js
Array(10) // Array mit 10 leeren Elementen
Array(10).fill(0) // Array mit 10 Nullen
Array(10).fill(0).map((_, i) => i) // Array mit Zahlen von 0 bis 9
// generell: <Array>.map((element, index, array) => ...)

Array.from({length: 10}, (_, i) => i) // Array mit Zahlen von 0 bis 9
Array.from(something) // Array aus Iterables (und ähnlichem)

const a = [1, 2, 3];
const [x, y, z] = a; // Array-Destrukturierung
const b = [0, ...a, 4]; // Entpacken
const c = [...a, ...a]; // Konkatenation
```

Objekte mit Funktionen


```js
const person = { // Objektliterale
    name: 'Alice',
    age: 30,
    greet: function() {
        return `Hallo, mein Name ist ${this.name}.`;
    },
    sayAge() { // Methodenkurzform (ES6)
        return `Ich bin ${this.age} Jahre alt.`;
    }
};
console.log(person.name, person['name']); // "Alice Alice"
person.greet();
const f = person["sayAge"]; // Funktionen sind Werte
f();
```

Arbeiten mit Objekten

```js
const keys = Object.keys(person);     // ['name', 'alter', ...]
const values = Object.values(person);       // ['Alice', 31, ...]
const entries = Object.entries(person);   // [['name', 'Alice'], ...]
// Spread-Operator
const flatCopy = {...person};             // Flache Kopie
const copyWithChanges = {...person, fähigkeiten: ['JS']};

// Objekt-Destrukturierung
const {name, alter} = person;                 // Variablen extrahieren
```

Funktionen erzeugen Objekte

Funktionen sind Objekte

- Funktionen sind *auch* Objekte, `f.x = 1` geht
- In Funktionen[3] gibt es:
  - `arguments` (alle Argumente)
  - `this` (Kontext, ein Objekt)

[3] Außer Arrow-Funktionen

Funktionen erzeugen Objekte

Funktionen sind Objekte

- Funktionen sind *auch* Objekte, `f.x = 1` geht
- In Funktionen[4] gibt es:
  - `arguments` (alle Argumente)
  - `this` (Kontext, ein Objekt)

Funktionen sind Konstruktoren

```js
function f() { this.x = 1; } // return ist implizit this
const o = new f(); // erstellt ein neues Objekt
console.log(o.x); // 1
```

[4] Außer Arrow-Funktionen

Prototypen

Objekte “erben” von anderen Objekten

- `obj.__proto__` verweist auf ein Prototyp-Objekt
- wenn kein Wert gefunden wird, wird im Prototyp gesucht

Prototypen

Objekte “erben” von anderen Objekten

- `obj.__proto__` verweist auf ein Prototyp-Objekt
- wenn kein Wert gefunden wird, wird im Prototyp gesucht

```js
const animal = { speak: () => `I'm a ${this.name}.` };
const dog = Object.create(animal); // dog.__proto__ = animal
dog.name = "dog";
dog.speak(); // "I'm a dog."
const cat = { name: "cat" };
Object.setPrototypeOf(cat, animal); // cat.__proto__ = animal
cat.speak(); // "I'm a cat."
```

Prototypen als Klassen mit Vererbung


```js
function Animal(name) { // Prototypen als Klassen
    this.name = name;
}
function Giraffe(name) { // Vererbung
    Animal.call(this, name); // Elternkonstruktor aufrufen
}
Giraffe.prototype = Object.create(Animal.prototype);
Giraffe.prototype.constructor = Giraffe; // Konstruktor zurücksetzen

const giraffe = new Giraffe('Gigi');
giraffe instanceof Giraffe; // true
giraffe instanceof Animal; // true
```

Vererbung ist nicht kompliziert, aber umständlich.

OOP: Klassen als syntaktischer Zucker über Prototypen (ES6+)


```js
class Animal { // Klassensyntax (Intern Prototypen)
    constructor(name) {
        this.name = name;
    }
    speak() {
        return `I'm a ${this.name}.`;
    }
}
class Giraffe extends Animal {
    constructor(name) {
        super(name); // Elternkonstruktor aufrufen
    }
}
const giraffe = new Giraffe('Gigi');
giraffe.speak(); // "I'm a Gigi."
```

Modul `math.js`

```js
export const PI = 3.1415;
export const add = (a, b) => a+b;
export default add;
```

Import in `main.js`

```js
import { PI, add } from './math.js';
console.log(PI); // 3.1415
console.log(add(2, 3)); // 5
```

```js
import add from "module-name"; // default Export
import { add } from "module-name"; // benannter Export
import * as name from "module-name"; // alle Exporte
```

CommonJS-Module exportieren, über `module.exports` und importieren mit `require()`:

```js
const PI = 3.1415;
const add = (a,b)=>a+b;
module.exports = {
  PI, // kurz für PI: PI
  add,
  default: add
};
```

```js
const math = require('./math.js');
const { PI, add } = require('./math.js');
const add = require('./math.js').default;
```

CommonJS-Module: synchronn keine Browserunterstützung.
CommonJS und ES-Module sind nicht (gut) kompatibel.

Fehlerbehandlung: Try/Catch/Finally


```js
function div(a, b) {
  if(b === 0) {
    throw new Error("Division durch Null");
  }
  return a / b;
}
function divDivideByZeroIsZero(a, b) {
  try {
    return div(a, b);
  } catch (e) { // fängt alle Fehler
    return 0; // e.message
  }
}
```

- Kernkonzept von JavaScript: *single-threaded* UND *asynchron*
- Viele JavaScript-Operationen sind asynchron
  - Dateien lesen, Netzwerkanfragen, Sleep, (DOM-)Events
- Asynchrone Operationen rufen immer Callbacks auf

Execution Model: Event Loop und Queue

- Job Queue: Enthält Funktionen (Callbacks)
- Execution Model: Runtime führt kontinuierlich Jobs aus
- Event Loop:
  - Es wird immer der erste Job in der Queue ausgeführt
  - Call Stack leer $\Longrightarrow$ nächster Job aus der Queue
  - Asynchrone Operationen werden in die Queue eingereiht

```js
setTimeout(() => { // sleep
    console.log('Die Zukunft ist da, Hurra!');
}, 1000); // 1000ms = 1s
```

Callbacks sind in der Praxis schnell unübersichtlich (“Callback-Hölle”):

```js
function doEverything(callback) {
  processA(data1 => {
    processB(data2 => {
      processC(data3 => {
        callback(data3);
      });
    })
  })
}
```

Promises: eine Abstrakton über Callbacks


```js
const promise = new Promise((resolve, reject) => {
    if (erfolg) { // asynchrone Operation
        resolve(result); // callback für Erfolg
    } else {
        reject(error); // callback für Fehlerfall
    }
});
promise.then(result => {
    // Erfolg
}).catch(error => {
    // Fehler
});
```

Operationen auf Promises


```js
getPromiseA
  .then(data => getPromiseB(data))
  .then(data => getPromiseC(data))
  .then(data => {
      // mache was mit data
  })
  .catch(error => {
      // Alle Fehler in der Kette landen hier
  });
```

- Promises lassen sich chainen (verkettete Aufrufe) und haben Fehlerbehandlung.
- Dabei werden `Promises<Promise<Data>>` zu `Promise<Data>` aufgelöst.
- Nützlich: `Promise.all([p1, p2, ...])` wartet auf mehrere Promises gleichzeitig

Async/Await (ES2017)

Async / Await ist eine syntaktische Abstraktion über Promises.

```js
promise.then(result => {
  /* mach was damit */ })
```

```js
const result = await promise;
/* mach was damit */
```

Innerhalb von `async`-Funktionen kann mit `await` gewartet werden.

```js
async function promiseAll() {
    const result1 = await getPromise1();
    const result2 = await getPromise2(result1);
    return result2;
}
```

Wichtig: Fehlerbehandlung nicht vergessen!

Wie funktioniert Async/Await?

- `async`-Funktionen geben immer ein Promise zurück
- `await` wartet auf das Ergebnis eines Promises
  - erstellt einen Callback für den Rest der Funktion
  - erstellt einen neuen Promise für diesen Callback
  - der Promise wird zurückgegeben
  - Aber: Nach außen sieht es aus wie eine synchrone Funktion!
- Fehlerbehandlung mit `try/catch`
- `await` kann nur
  - innerhalb von `async`-Funktionen verwendet werden
  - teilweise auf dem Top-Level (ES2022)

Async/Await: Beispiel


```js
async function fetchData() {
    try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Fehler:', error);
    }
}
fetchData();
```

JavaScript-Ökosystem

Standard Bibliothek

- JavaScript hat keine Standardbibliothek im klassischen Sinne
- Es gibt viele eingebaute Objekte und Funktionen
  - `Math`, `Date`, `JSON`, `Promise`, `Set`, `Map`, `Error`
- Es gibt globale Objekte
  - `console`: Ausgabe
  - `window`: globales `this` in Browser-Umgebungen
  - `global`: globales `this` in Node.js
  - `globalThis`: globale Umgebung (einheitlich seit ES2020)
- Darüber hinaus gibt es je nach Runtime verschiedene globale Objekte:
  - `fs`: Dateisystem (Node.js)
  - `fetch`: HTTP-Client (Browser, Node.js)
  - `window`: beinhaltet Browser-APIs
  - `window.document`: Document Object Model (Browser)

Typisches Problem, vor allem in Webanwendungen:

Daten zwischen Client und Server übertragen:

1. Serialisieren, z.B. zu einem String
2. Übertragen, z.B. über HTTP / WebSocket / TCP
3. Deserialisieren d.h. Parsen bzw. wieder Einlesen

[5] Ein JavaScript-Entwickler und -Advokat

[6] `eval` ist problematisch, weil es Code ausführen kann - es gibt inzwischen Besseres!

Typisches Problem, vor allem in Webanwendungen:

Daten zwischen Client und Server übertragen:

1. Serialisieren, z.B. zu einem String
2. Übertragen, z.B. über HTTP / WebSocket / TCP
3. Deserialisieren d.h. Parsen bzw. wieder Einlesen

- Erkenntnis von Douglas Crockford[7]:
  - Eine JavaScript Teilmenge ist dafür gut geeignet
  - JavaScript hatte bereits `eval` zum Parsen[8]
  - JSON = JavaScript Object Notation

JSON ist das de facto Standardformat für Datenserialisierung überall geworden.

---

[7] Ein JavaScript-Entwickler und -Advokat

[8] `eval` ist problematisch, weil es Code ausführen kann - es gibt inzwischen Besseres!

```js
const jsonString = '{"name": "Alice", "age": 30}';
const jsonObject = JSON.parse(jsonString);
jsonObject; // { name: "Alice", age: 30 }
jsonObject.name; // "Alice"
const newJsonString = JSON.stringify(jsonObject);
newJsonString; // '{"name":"Alice","age":30}'
```

```js
const jsonString = '{"name": "Alice", "age": 30}';
const jsonObject = JSON.parse(jsonString);
jsonObject; // { name: "Alice", age: 30 }
jsonObject.name; // "Alice"
const newJsonString = JSON.stringify(jsonObject);
newJsonString; // '{"name":"Alice","age":30}'
```

Unterstützte Datentypen:

- Objekte: `{ "key": value, ... }`
- Arrays: `[value, ...]`
- Strings: `"string"`
- Zahlen: `123`, `-123.45`, `0.1e10`
- Booleans: `true`, `false`
- Null: `null`

- Andere Typen werden umgewandelt.
- JSON unterstützt keine Kommentare.
- JSON verlangt doppelte Anführungszeichen `"`:
  - für Schlüssel von Objekten
    kein `{ age: 3 }`
  - für Strings
- erlaubt keine trailing commas `[1, 2, 3,]`

TypeScript ergänzt JavaScript um ein Typensystem.

- Überwiegend eine Obermenge von JavaScript.
- Etwas stärkere Unterstützung für OOP.
- Kompiliert zu JavaScript: Typen werden entfernt.
- Statisch typisiert, automatische Typinferenz.
- Typensystem wesentlich flexibler als Java / Python.
- Gradual Typing d.h. JavaScript und TypeScript mischen.

Mehr zu TypeScript im nächsten Teil!

Definitionen	
Engine	Software, die JavaScript-Code ausführt
Runtime	Umgebung, in der JavaScript-Code ausgeführt wird
Paketmanager	Tool, das Pakete verwaltet und installiert
Bundler	Tool, das JavaScript-Module packt und optimiert
Transpiler	Tool, dass Code in eine wünschenswerte Form bringt
Linter	Tool, das Code auf Stil und Fehler überprüft
Bibliothek	Sammlung von wiederverwendbarem Code
Framework	Bibliothek, die Struktur und Konventionen vorgibt

- `npm` ist der Standard-Paketmanager (Node Package Manager)
  - CLI-Tool: verwaltet Pakete[9] und Abhängigkeiten
  - `> npm install` installiert Pakete ins Verzeichnis `node_modules`
  - `package.json` enthält die benötigten Pakete und Metadaten
  - `node_modules/` enthält Pakete und deren Abhängigkeiten (als Unterordner)
  - `package-lock.json` enthält die genauen Paketversionen
- Paketdatenbank unter https://www.npmjs.com/
- Es gibt alternative CLI-Tools, z.B. `yarn`, `pnpm`, `bun`

[9] Lokale Projektordner sind hier auch Pakete

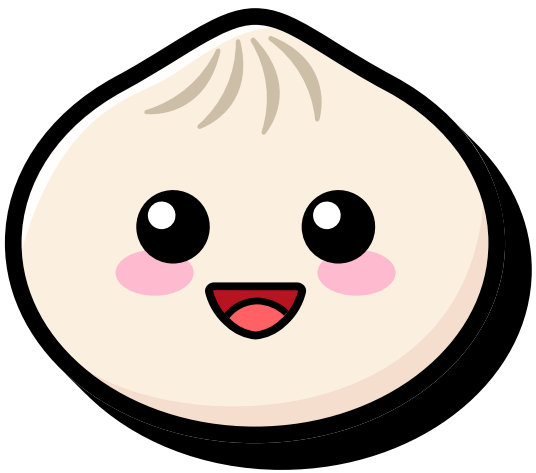
`npm init` initialisiert ein neues Projekt und erstellt eine `package.json`-Datei:

```json
{
  "name": "demo-package",
  "version": "0.1.0",
  "author": "Lorenz",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "lodash": "^4.17.21"
  },
  "type": "module"
}
```

JavaScript Ökosystem: Bundler

- Bundler sind Tools, die JavaScript-Module bündeln und optimieren
- Sie helfen, den Code für eine Produktivumgebung vorzubereiten
- Beispiele:
  - `webpack`
  - `rollup`
  - `esbuild`
  - `vite`
- Bundler sind oft auch Transpiler und Linter
- Klassischerweise braucht man für TypeScript einen Transpiler oder Bundler

One tool to rule (them all)



Bun

- `bun` ist
  - ein JavaScript-Runtime, kompatibel zu Node.js.
  - ein Paketmanager, kompatibel zu `npm`.
  - ein Bundler.
  - ein Test-Runner.
  - kann TypeScript direkt ausführen.
  - kann ES-Module und CommonJS-Syntax mischen.
- `bun` hat eigene Standardbibliothek
  - Datenbankanbindung
  - HTTP-Server mit Hot-Reload (d.h. live aktualisieren)

TypeScript

learnxinyminutes.com/
typescript

- TypeScript passiert zur Compile-Zeit (Type Checker)
- TypeScript-Compiler produziert JavaScript-Code (Transpier)
- TypeScript kann JavaScript-Code analysieren. Das bringt uns:
  - Bessere Editor-Unterstützung (IntelliSense, Autocomplete, …)
  - Viele Typen werden automatisch erkannt (automatische Typinferenz)
  - Vermeidung viele “dummer Fehler”

```ts
let name: string = 'Alice';
let age: number = 30;
let isStudent: boolean = true;
// Arrays:
let hobbies: string[] = ['reading', 'sports'];
// Tuples:
let bookAndPageNumber: [string, number] = ['JS Basics', 42];
let person: { name: string; age: number } = { name: 'Alice', age: 30 };
// Funktionen:
let greet: (name: string) => string = (name) => `Hallo, ${name}!`;
```

```ts
// Union Types
let value: string | number;
value = 'Hallo';
value = 42;

// Intersection Types
type Person = { name: string; age: number };
type Employee = { id: number; position: string };
type EmployeePerson = Person & Employee;
```

TypeScript: Interfaces


```ts
interface Person {
  name: string;
  age: number;
  greet: (name: string) => void;
}
```

```ts
const person: Person = {
  name: 'Alice',
  age: 30,
  greet: (name) => { },
};
```

- Interfaces sind eine Möglichkeit, Typen zu definieren.
  - beschreiben die Struktur eines Objekts
  - können erweitert werden
- Sie können auch Methoden und Eigenschaften definieren.

```ts
let value: any = 'Hallo'; // Any-Typ
value = 42; // kein Fehler

const person: { name?: string } = {}; // optionales Attribut

const p: Promise<number>; // Promise mit Typ
const f = async (): Promise<number> => 3;

const f2 = <T>(x: T): T[] => { return [x]; }; // Generics
const f3 = <T extends Person>(x: T): void => x.greet('Hallo');
```

```ts
type Status = { type: "loading" }
            | { type: "success", value: any }
            | { type: "error", message: string };

let status: Status = { type: "loading" };
function updateStatus(newStatus: Status) { // State transition
    if(status.type !== "loading")
        throw new Error("Cannot update non-loading status");
    status = newStatus;
}
```

Solche Patterns können sehr gut wartbaren Code erzeugen:

- `Status`-Werte sind quasi selbsterklärend.
- Der Zustand und Änderungen daran sind wohldefiniert und übersichtlich.

Browser-APIs

Browser-APIs

Es gibt viele APIs, die in Browsern verfügbar sind:

- DOM (Document Object Model)
- BOM (Browser Object Model)
- Web APIs
- WebAssembly
- WebSockets
- Fetch API
- Geolocation API
- Web Storage API
- IndexedDB API

```js
const element = document.getElementById('myElement');
document.getElementsByClassName('myClass'); // getElementsByTagName
element.innerHTML = '<p>Hallo, Welt!</p>';
element.innerText = 'Hallo, Welt!';
element.style.color = 'red';
element.children; // HTMLCollection, Array-ähnlich
element.id; // 'myElement', HTML-Attribute manipulieren
element.addEventListener('click', () => {
    console.log('Element wurde geklickt!');
});
const newElement = document.createElement('div');
element.appendChild(newElement);
document.cookie; // Cookies lesen
```

```js
window.alert('Hallo, Welt!'); // Alert-Dialog
window.location.href = 'https://example.com'; // neu Seite laden
window.history.back(); // zurück zur vorherigen Seite
window.decodeURIComponent('Hallo%20Welt'); // URL dekodieren
window.c
```

```js
// Local Storage bleibt auch nach Schließen des Browsers erhalten
window.localStorage.setItem('key', 'value');
const value = window.localStorage.getItem('key');
// Session Storage bleibt nur für die aktuelle Sitzung erhalten
window.sessionStorage.setItem('key', 'value');
const value = window.sessionStorage.getItem('key');
```

Zusammenfassung

- JavaScript ist eine vielseitige Sprache, die in vielen Bereichen eingesetzt wird.
- Es gibt viele Tools und Frameworks, die die Entwicklung erleichtern.
- TypeScript ist eine beliebte Erweiterung von JavaScript mit Typisierung.
- Browser-APIs ermöglichen den Zugriff auf viele Funktionen des Browsers.