

Übung 03: JavaScript

- ☒ Statt weniger großer Aufgaben empfehlen sich zum Programmieren Lernen eher viele kleinere Aufgaben.

Achte deshalb darauf, dass du nicht zu viel Zeit mit einer Aufgabe verbringst.

Aufgabe 0.1 node.js (10 min)

Installiere Node.js und npm (Node Package Manager) auf deinem Rechner.

Mit Admin-Rechten

```
1 # Alte Paketversion von Ubuntu entfernen
2 sudo apt-get remove nodejs
3 # Installiere die neueste Version von Node.js
4 curl -fsSL https://deb.nodesource.com/setup_22.x | sudo -E bash -
5 sudo apt-get install -y nodejs
```

Ohne Admin-Rechte

Die original Anleitung findest du hier: <https://github.com/nvm-sh/nvm?tab=readme-ov-file#installing-and-updating>

```
1 # Installiere nvm (Node Version Manager)
2 curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash
3 source ~/.config/nvm/nvm.sh
4 nvm install node
```

Aufgabe 0.2 - Bun (2 min)

Installiere Bun (<https://bun.sh/>):

```
1 curl -fsSL https://bun.sh/install | bash
```

Aufgabe 1 - Einfache JavaScript-Programme (25 min)

1. Schreibe ein JavaScript-Programm, das den Text „Hello World“ in der Konsole ausgibt.
2. Schreibe ein JavaScript-Programm, dass die ersten 10 Quadratzahlen in einer **for**-Schleife berechnet und in der Konsole ausgibt.
3. Schreibe ein JavaScript-Programm, dass die ersten 10 Quadratzahlen in einer **for**-Schleife berechnet und ausgibt, indem sie die **ersten n ungeraden Zahlen aufsummiert**.
4. Schreibe eine JavaScript-Funktion, die die ersten n Quadratzahlen berechnet und als Array zurück gibt.
5. Schreibe eine JavaScript-Funktion, in der ein Array der ersten n Zahlen erstellt wird (siehe Folie zu „Array-Syntax und -Methoden“ (26)). Berechne folgende Werte aus dem Array:
 1. die Summe der Zahlen mit einer **for-of**-Schleife
 2. die Summe der Zahlen mit **.forEach**
 3. die Summe der Zahlen mit einer **.reduce**-Funktion (ohne eine **let**-Variable)
 4. die ersten n Quadratzahlen mit **.map**

Aufgabe 2 - Einfache HTML-Seite generieren lassen (20 min)

Schreibe ein TypeScript-Programm, dass mithilfe von Template-Strings eine HTML-Seite für einen Steckbrief (Vorstellung einer Person) erstellt. Erstelle dafür zunächst ein *einfaches* Datenmodell¹ für eine Person und erstelle dann eine Funktion, die eine HTML-Seite (als String) für diese Person erstellt.

Du kannst TypeScript-Dateien mit bun ausführen. Erstelle dafür eine Datei (z.B. `index.ts`) und führe sie mit dem Befehl `bun index.ts` aus. Du kannst auch den Befehl `bun run index.ts` verwenden, um die Datei auszuführen.

Erstelle anschließend einen eigenen Typen für eine Person und darauf aufbauend einen Typen für mehrere Personen. Erstelle jeweils eine HTML-Seite für jede Person und speichere sie in einer Datei `<name>.html` ab. Du kannst dafür die `fs`-Bibliothek verwenden, die node und bun mitbringen:

```
1 import fs from "fs";
2 const html = `<!DOCTYPE html> ...`;
3 fs.writeFileSync("index.html", html);
```

⚠ Vorsicht: Dieser Code ist synchron, d.h. er blockiert den Event-Loop, bis die Datei geschrieben ist. In der Praxis solltest du asynchrone Funktionen verwenden, um die Performance zu verbessern.

¹Es reichen eine kleine Hand voll Angaben und die Seite muss nicht schön gemacht werden - das kostet sonst nur wertvolle Zeit!

Aufgabe 3 - Asynchrones JavaScript/TypeScript (25 min)

Das folgende Code-Snippet verzögert die Ausgabe von „Hello World!“ um 1 Sekunde.

```
1 const delay = (time) => new Promise(resolve => setTimeout(resolve, time));
2 delay(1000).then(() => console.log("Hello World!"));
```

js

1. Schreibe eine Funktion `squareVoid(n: number): void`, die eine Zahl als Parameter entgegennimmt, die Zahl sofort ausgibt und mit 1 Sekunde Verzögerung dessen Quadratzahl ausgibt. Du kannst `.then` verwenden, um die Verzögerung zu realisieren.
2. Schreibe eine Funktion `squarePromise(n: number): Promise<number>`, die eine Zahl als Parameter entgegennimmt, die Zahl sofort ausgibt und mit 1 Sekunde Verzögerung dessen Quadratzahl zurückgibt. Schreibe diese Funktion in den folgenden zwei Varianten:
 1. als `async function`, die `await` auf einen Aufruf an `delay` anwendet. Tipp: Der Return-Wert von `async`-Funktionen ist immer ein `Promise` über dem Return-Wert der Funktion.
 2. als reguläre Funktion, die `.then` auf `delay` aufruft. Tipp: Das Ergebnis von `.then` ist immer ein neuer `Promise` über dem Return-Wert der Funktion in `.then`.
3. Schreibe eine Funktion `delayTimes(n: number, time: number): Promise<number>`, die mehrere `delay`-Aufrufe hintereinander ausführt, bevor sie das Ergebnis zurück gibt.
4. Erstelle die folgenden Funktionen, die mit mehreren Zahlen arbeiten:
 1. Erstelle eine Funktion `squareMultiple(nums: number[]): Promise<number>[]`, die eine Liste von Zahlen als Parameter entgegennimmt und für jede Zahl die Funktion `squarePromise` aufruft. Die Funktion gibt eine Liste von Promises zurück, die die Ergebnisse der Berechnungen enthalten. Rufe die Funktion mit der globalen Funktion `Promise.all` auf, um eine Liste von Promises `Promise<number>[]` in ein Promise von einer Liste `Promise<number[]>` zu verwandeln. Du kannst dann die Ergebnisse mit `.then` oder `await` abfragen.
 2. Erstelle eine `async` Funktion `squareMultipleAsyncAwait(nums: number[]): Promise<number[]>`, die eine Liste von Zahlen als Parameter entgegennimmt in einer `for`-Schleife für jede Zahl die Funktion `squarePromise` mit `await` aufruft und in eine neue Liste speichert. Warum ist die vorherige Lösung mit `Promise.all` besser als diese Lösung?

Aufgabe 4 - Schiffe versenken (30 min)

☒ Diese Aufgabe ist auch ein Einstieg/Übergang zur Einheit „Frontend“.

Programmiere ein minimalistisches Schiffe-Versenken-Spiel in JavaScript und HTML. Das ist viel einfacher als es zunächst klingt - du kannst gerne in einer einzelnen HTML-Datei mit eingebettetem JavaScript arbeiten.

- Es gibt nur einen Spieler, der gegen den Computer spielt.
- Dein Spielfeld ist folgendes 5x5 Raster mit 0 (Wasser) und 1 (Schiff):

```
1 const field = [[0, 1, 1, 1, 0],  
2               [0, 0, 0, 0, 1],  
3               [1, 0, 0, 0, 1],  
4               [1, 0, 1, 0, 1],  
5               [0, 0, 0, 0, 1]];
```

js

- Das Spielfeld wird als 5-column Grid mit dargestellt, dass (erstmal leere) Buttons enthält.
 - Wenn der Spieler auf einen Button klickt, wird eine Funktion, z.B. `shoot(row, col)` aufgerufen (siehe unten).
 - Generiere die Buttons per JavaScript aus dem Spielfeld und füge sie mit den DOM-Methoden wie z.B. `.innerHTML` oder `.appendChild` auf dem Feld-Grid-Element `document.getElementById("idOffFieldElement")` ein.
 - Benenne die Buttons mit `id`-Attributen, die die Zeilen- und Spaltennummern enthalten (z.B. `id="cell_0_0"`), damit du sie über `document.getElementById` leicht ansprechen kannst.
- Zusätzlich zum Spielfeld gibt es zwei Elemente, die anzeigen wie viele Schiffe getroffen wurden und wie viele Schüsse insgesamt abgegeben wurden.
- Die Funktion `shoot(row, col)` kann sehr einfach sein:
 - Wenn der Spieler auf ein leeres Feld klickt, wird der Button blau gefärbt und erhält die Beschriftung 0.
 - Wenn der Spieler auf ein Feld mit einem Schiff klickt, wird der Button rot gefärbt und erhält die Beschriftung X.
 - Die beiden Counter für Schüsse und Treffer werden aktualisiert. Du kannst diese Elemente als globale Variablen definieren oder sie analog zu den Vorlesungsfolien mit einem Objekt kapseln (Suchbegriff „Closure“).
 - Sobald der Spieler alle 10 Schiff-Felder getroffen hat, wird eine Nachricht ausgegeben, dass das Spiel gewonnen wurde.