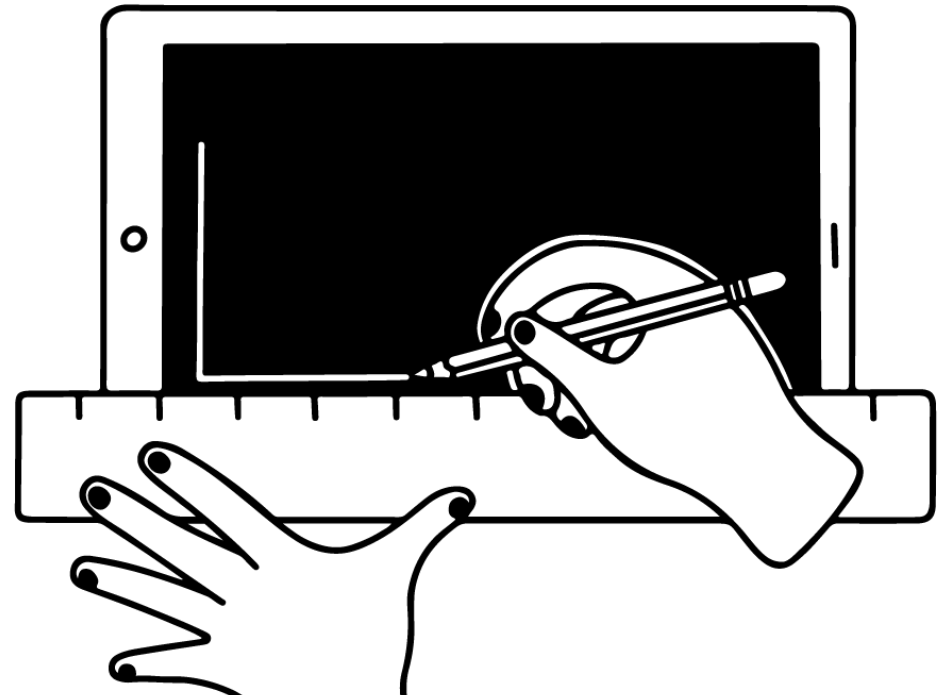
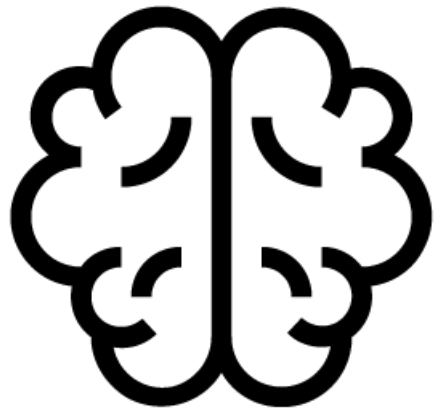


TypeScript Crash Kurs



Louis Wilke

exeta



Agenda

Setup (00)

Seite 3

Basics (01)

Seite 5

Übung 1: Hello World

Seite 13

Advanced (02)

Seite 14

Übung 2: Async Google Ping Tester

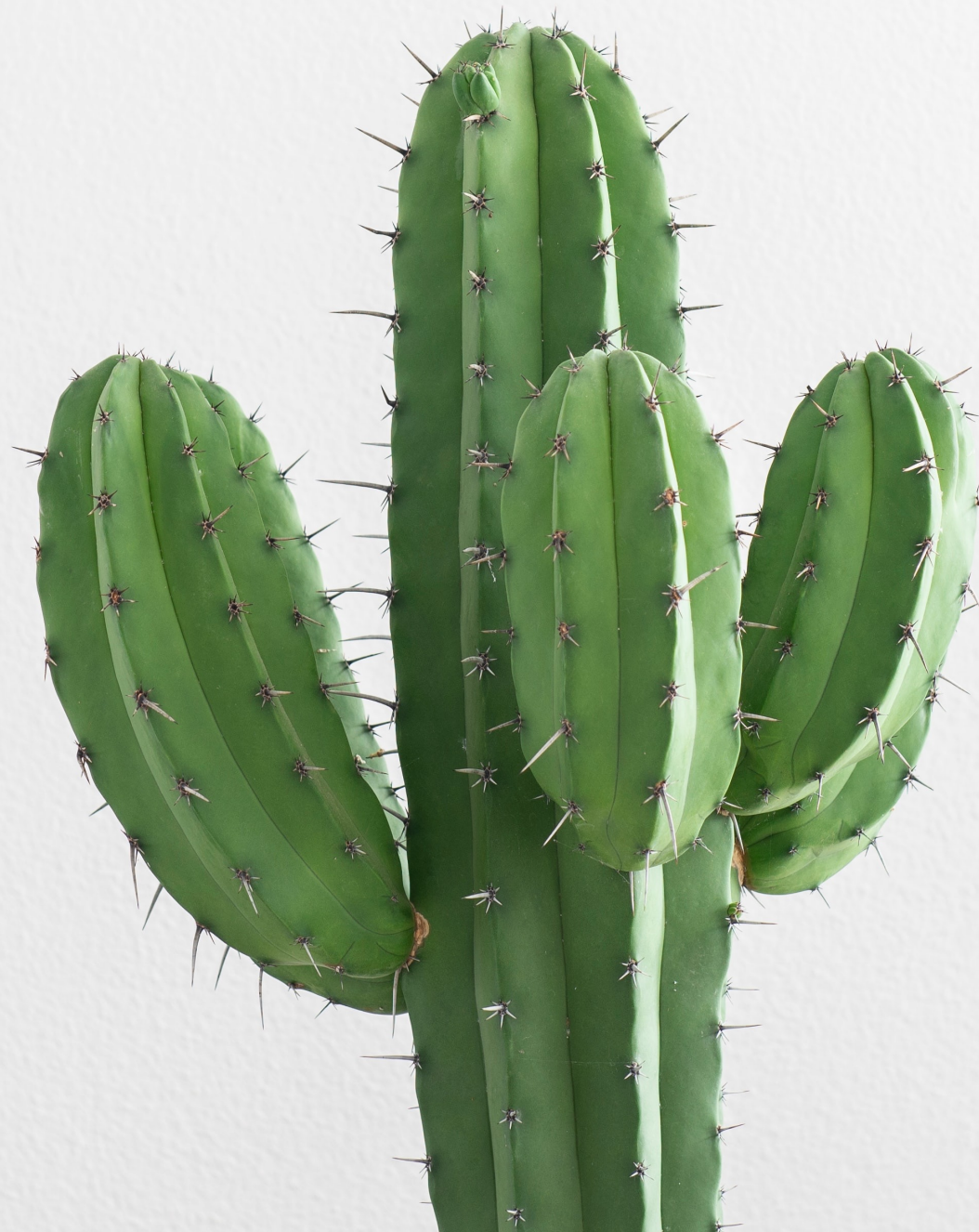
Seite 22

Übung 3: Webserver

Seite 23

00

Setup



Installation & Setup

NVM installieren:

- Unix: <https://github.com/nvm-sh/nvm>
- Windows: <https://github.com/coreybutler/nvm-windows>

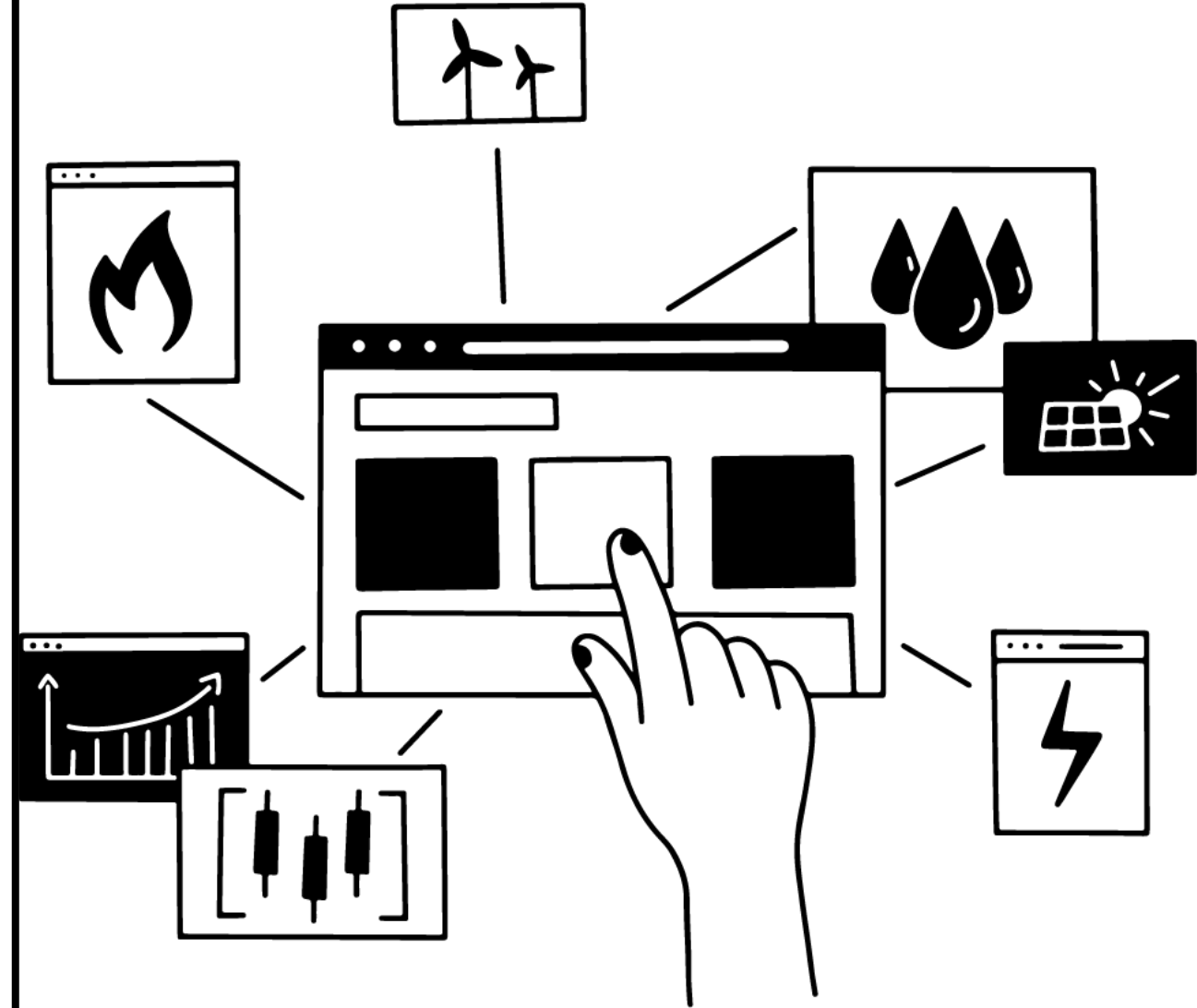
NodeJS Version 20 installieren: `nvm install 20`

In den Übungen (dort, wo die `package.json` liegt):

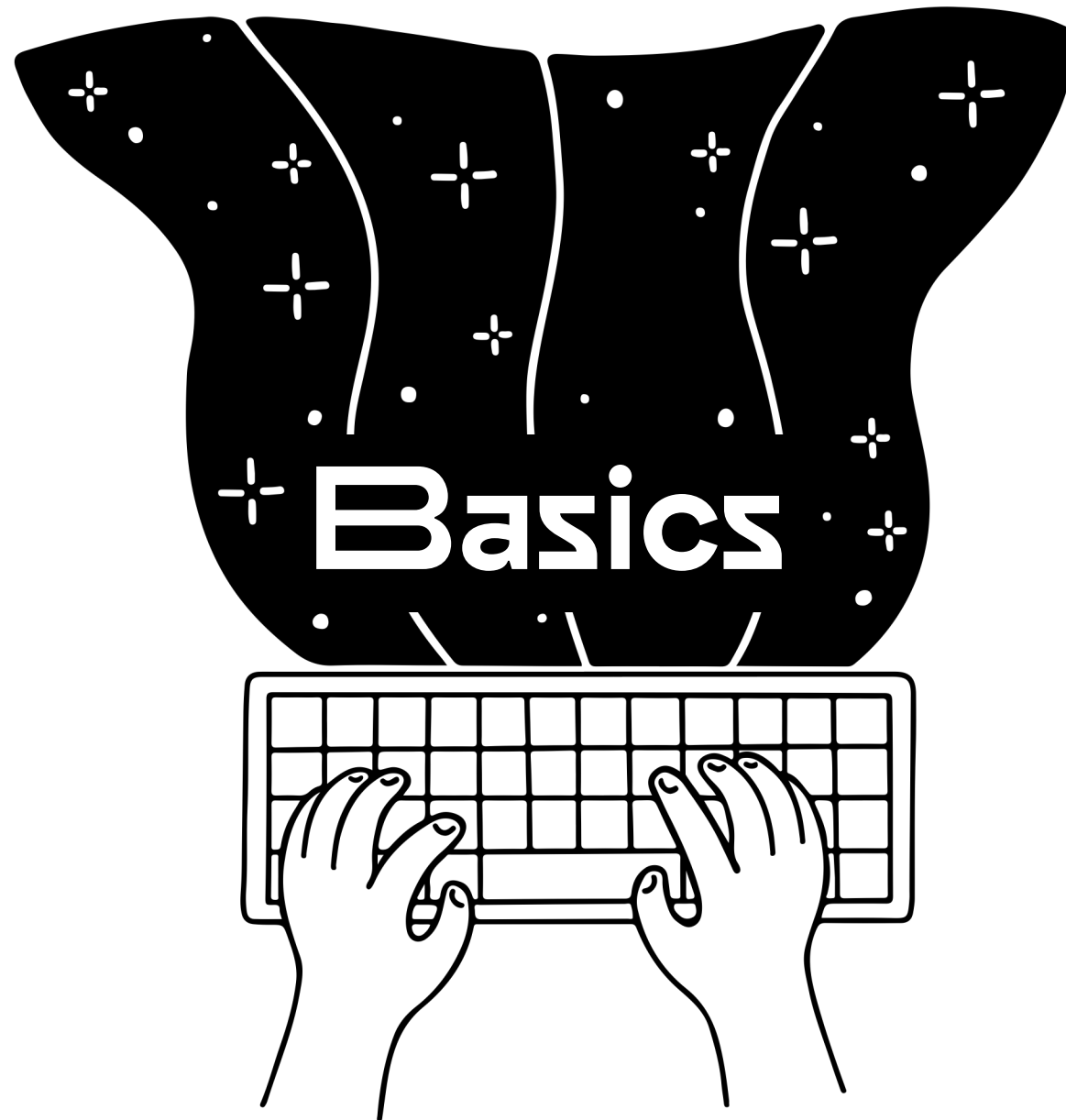
- Dependencies installieren: `npm i` oder `npm install`
- Script starten: `npm start` oder `ts-node index.ts`

Exercises, solutions, and slides on GitHub:

<https://github.com/SoulKa/typescript-tutorial>



01



Datentypen

Typ	string	number	boolean	null	undefined	symbol	object (& any)
Beispiel	<code>"exxeta"</code> <code>""</code> <code>'DDS'</code> <code>`Multiline string`</code>	<code>1e3</code> <code>-15</code> <code>NaN</code> <code>Number.POSITIVE_INFINITY</code>	<code>true</code> <code>false</code>	<code>null</code>	<code>undefined</code>	<code>new Symbol("hi")</code>	<pre>{ field: 123 }</pre>
Besonderheit	Keine Klasse wie in Java, sondern ein primitiver Typ	<p>Sowohl Floats als auch Integer.</p> <p>NaN stellt eine ungültige Zahl dar.</p> <p>Für extrem große Integer gibt es <code>bigint</code>.</p>		Kann nur dem Typ <code>null</code> zugewiesen werden. Eine Variable mit einem anderen Datentyp kann nicht <code>null</code> sein.	<code>undefined == null</code> ist <code>true</code> <code>undefined === null</code> ist <code>false</code>	<p>Habe ich tatsächlich noch nie benutzen müssen. Ist immer einzigartig:</p> <pre>new Symbol("hi") === new Symbol("hi") ist false</pre>	<p>Ein POJO oder eine Klassen Instanz.</p> <p><code>any</code> ist ein <code>object</code> oder einer der primitiven Typen.</p>

Spezialfälle:

- `void`: Für Methoden ohne Rückgabewert, kennen wir aus Java.
- `never`: Für nicht erreichbaren Code, z.B. wenn in den Zeilen davor ein Error geworfen wird.
- `unknown`: Wie `any`, aber man kann mit `unknown` nicht weiterarbeiten.

Variablen und Strings

- Variablen werden durch `const`, `let` oder `var` deklariert. **Nutzt niemals `var`!** Die buggen euch im globalen Kontext rum.
- Beliebige Objekte können mit `+` mit Strings konkateniert werden
- Template Literals sind mit ``` gekennzeichnet und können durch `${myVar}` Variablen oder ganze Methodenaufrufe einfügen
- Arrays können durch `.join()` zu einem einzelnen String verbunden werden

```
const name : "World" = "World";  
const str : string = "Hello " + name; // Concatenation  
let str2 : string = `Hello ${name}`; // Template literal  
var str3 : string = ["Hello", name].join(" "); // Array join
```

Funktionen Definieren

Mit `function` keyword



```
1 function hello() : void {  
2   console.log("Hello World");  
3 }
```

`hello()` ist eine normale Funktion.

Als Lambda in Variable



```
1 const hello = () : void => {  
2   console.log("Hello World");  
3 }
```

`hello()` ist eine Konstante Variable welcher eine Funktion zugewiesen ist. Die Funktion ist durch ein Lambda beschrieben.

Klassenmethode



```
1 class Test {  
  1 usage  
2   hello() : void {  
3     console.log("Hello World");  
4   }  
5 }
```

`hello()` ist eine Methode der Klasse `Test`.

Default Parameter

Default Parameter mit = zuweisen

```
function withDefaultParameter(name : string = 'World') : void {  
    console.log(`Hello ${name}`)  
}
```

Optionaler Parameter mit ? kennzeichnen

```
function withOptionalParameter(name?: string) : void {  
    if (name) {  
        console.log(`Hello ${name}`)  
    } else {  
        console.log('Hello World')  
    }  
}
```

Package Manager

NPM

NPM ist der Package Manager von JavaScript + TypeScript Modulen.

Dependency (JavaScript) installieren:

```
npm install express
```

Dev-Dependency (z.B. TypeScript Definitionen) installieren:

```
npm install -d @types/express
```

Immer open-source und meist mit guter Doku. Alles auf einen Blick.

NPM modifiziert die `package.json`.

express

DT

4.18.2 • Public • Published a year ago

Readme

Code Beta

31 Dependencies

74.335 Dependents

270 Versions

express

Fast, unopinionated, minimalist web framework for **Node.js**.

npm

v4.18.2

install size 1.89 MB

downloads 125.3M/month

```
const express = require('express')
const app = express()

app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

Installation

This is a **Node.js** module available through the **npm registry**.

Before installing, **download and install Node.js**. Node.js 0.10 or higher is required.

If this is a brand new project, make sure to create a `package.json` first with the **npm init** **command**.

Installation is done using the **npm install** **command**:

\$ npm install express

Follow **our installing guide** for more information.

Install

> npm i express

Repository

github.com/expressjs/express

Homepage

expressjs.com/

Weekly Downloads

28.983.593

Version

4.18.2

License

MIT

Unpacked Size

214 kB

Total Files

16

Issues

130



Pull Requests

61

Last publish

a year ago

Collaborators

> Try on RunKit

Report malware

Hello World

Hello World Programm in TypeScript:

```
1 console.log("Hello World");
```

Type**Script** ist eine Script-Sprache. Das Programm läuft einfach in Zeile 1 los.

Wenn man es als ordentliche Methode wie in Java haben möchte:

```
1 /**  
2  📢 The main function.  
3  * @param args The command line arguments.  
4  */  
5 1 usage  
6 function main(args: string[]): void {  
7     console.log("Hello World");  
8 }  
9 main(process.argv.slice(2));
```



Imports und Exports

- Es gibt genau einen `default` Export pro Datei
- Man kann Klassen, Funktionen, oder Variablen exportieren
- Exportiert wird entweder direkt bei der Definition oder später in der Datei (z.B. am Ende)
- In TypeScript ist es okay mehrere Dinge in einer Datei zu definieren und exportieren

- Importiert wird immer ganz oben
- Default import ohne, alle anderen Imports mit Klammern

```
1  export default class RandomClass {                                random-class.ts
2      1 usage
3      getRandomNumber() : number {
4          |   return Math.random();
5      }
6  }
7
8      2 usages
9  export const PROJECT_NAME : "TypeScript Workshop" = "TypeScript Workshop";
```

```
1  import RandomClass, {PROJECT_NAME} from "./exports";
2
3  const randomClass : RandomClass = new RandomClass();
4  console.log(randomClass.getRandomNumber());
5  console.log(PROJECT_NAME);
```

Übung 1 – Hello World

15 Minuten

Modifiziert die `index.ts`

1. Implementiert eine Methode `sayHello(name: string)`, welche einen `string` nimmt und ihn auf die Konsole loggt
2. Importiert `getRandomName()` aus `src/random-name-generator.ts`
3. Macht den `name` Parameter optional und nutzt `getRandomName()` um dann einen zufälligen Namen zu nutzen
4. Implementiert dieselbe Methode als Lambda und speichert sie in der Konstante `sayHelloLamda` ab.

Details in der `README.md` und in den TODOs im Code



**May the source code
be with you**

02

Advanced Syntax



Klassen

- Es gibt nur eine Constructor Implementierung! Man kann aber mehrere Overloads definieren.
- Man kann die Sichtbarkeit von Feldern auch direkt im Constructor angeben. Damit entfällt das explizite setzen im Constructor und die Feld Definition
- Getter und Setter sind keine normalen Methoden. Man nutzt `get` und `set`.
- Es gibt keine Packages in TypeScript. Wo eure Klassen, Typen und Interfaces liegen ist also quasi egal.
Aber: es gibt namespace!

```
1 class MyClass {  
2  
3     third: number  
4  
5     1 usage  
6     public constructor(private readonly first: number, public second: number, third : string = "0") {  
7         this.third = Number.parseInt(third);  
8     }  
9  
10    // getter  
11    1 usage  
12    public get valuesAsString(): string {  
13        return this.first + ", " + this.second + ", " + this.third;  
14    }  
15  
16    }  
17  
18    console.log(new MyClass( first: 1, second: 2, third: "3").valuesAsString);
```

Interfaces

Interfaces (und auch Types) sind **keine Klassen** und **existieren zur Laufzeit** nicht. Sie werden nur zum Kompilieren genutzt

→ instanceof geht hier nicht, da zur Laufzeit geprüft

```

1  declare interface User {
2      name: string;
3  }
4
5  declare interface UserWithAge extends User {
6      age: number;
7
8      isAdult(): boolean;
9  }
10
11 class UserImpl implements User, UserWithAge {
12     constructor(public name: string, public age: number) {
13     }
14
15     sayHello(): void {
16         console.log(`Hello ${this.name} (${this.age})`);
17     }
18
19     isAdult(): boolean {
20         return this.age > 18;
21     }
22 }
23
24 const user : UserImpl = new UserImpl( name: 'Marc', age: 42);

```


Types

Types können genau wie Interfaces von Klassen implementiert werden, ist aber eher unüblich.

Sie können kombiniert werden (&), oder separat vorkommen (|).

```

1  declare enum EventType {
2      User = 'USER',
3      Data = 'DATA',
4  }
5
6  declare type User = {
7      type: EventType.User;
8      name: string;
9  };
10
11  declare type UserWithAge = User & {
12      age: number;
13  };
14
15  declare type Data = {
16      type: EventType.Data;
17      value: number | string;
18  };
19
20  declare type Event = {
21      timestamp: number;
22      payload: User | UserWithAge | Data;
23  };
24
25  const event: Event = {
26      timestamp: Date.now(),
27      payload: {
28          type: EventType.User,
29          name: 'Marc',
30          age: 42,
31      }
32  }
33
34  if (event.payload.type === EventType.User) {
35      console.log(event.payload.name);
36  }

```

IntelliJ kann den Typ des Event Payloads durch den Vergleich herleiten.

Async/Await

`async` macht eine normale Methode zu einer asynchronen. Diese können dann im Hintergrund parallel laufen.

Man kann nur in asynchronen Methoden `await` benutzen.

`await` macht asynchrone Aufrufe wieder sequentiell.

`Promise.all(Promise<T>[]) : T[]`
wartet auf alle asynchronen Aufrufe parallel.

`Promise.race(Promise<T>[]) : T`
wartet auf den ersten Aufruf, der fertig wird.

```

4  const FILES : string[] = ["test-file.txt", "file-2.txt", "file-3.txt"];
5
6  1 usage  ⚡ Wilke, Louis
7  function printFileSync(filename: string) : void {
8      console.log(readFileSync(filename, options: "utf-8"));
9  }
10
11  2 usages  ⚡ Wilke, Louis
12  async function printFile(filename: string) : Promise<void> {
13      console.log(await readFile(filename, options: "utf-8"));
14  }
15
16  1 usage  ⚡ Wilke, Louis
17  async function printAllFiles() : Promise<void> {
18      for (const file : string of FILES) {
19          await printFile(file);
20      }
21  }
22
23  1 usage  ⚡ Wilke, Louis
24  async function printAllFilesParallel() : Promise<void> {
25      const promises : Promise<string>[] = FILES.map(file : string => readFile(file, options: "utf-8"));
26      const contents : string[] = await Promise.all(promises);
27      contents.forEach(content : string => console.log(content));
28  }

```

Promises

Promises sind die CompletableFuture von TS.

Eine `async` Methode gibt immer eine Promise zurück. Der return Wert wird intern durch `resolve()` gesetzt.

Errors werden durch `reject()` gesetzt.

In modernem TS müssen wir Promises kaum direkt erstellen.

```

1  async function getNumberAsync() : Promise<number> {
2      return 1;
3  }
4
1 usage  👤 Wilke, Louis
5  function getNumberPromise() : Promise<number> {
6      return new Promise<number>(
7          executor: (resolve, reject) : void => {
8              resolve(value: 1);
9          }
10     );
11 }
12
1 usage  👤 Wilke, Louis
13 function getNumberPromiseResolve() : Promise<number> {
14     return Promise.resolve(value: 1);
15 }
16
1 usage  👤 Wilke, Louis *
17 async function main() : Promise<void> {
18     console.log(await getNumberAsync());
19     getNumberPromise().then(number : number => console.log(number)).catch(console.error);
20     console.log(await getNumberPromiseResolve());
21 }
22
23 main().catch(console.error);

```

Generics

Man kann generic Parameter beliebig als Typ verwenden. Dabei kann man ihn auf typen beschränken (`extends`).

Bei Klassen ist der generische Typ in der gesamten Klasse verfügbar.

Man kann auch hier einen default setzen, oder Typen verodern.

Die generic Parameter kann man ad-hoc, oder mit existierenden Typen befüllen.

```

1  function parseJson<T extends object>(json: string) :T {
2      return JSON.parse(json) as T;
3  }
4
5  1 usage  👤 Wilke, Louis *
   class JsonParser<T extends (object | null) = object> {
6      1 usage  👤 Wilke, Louis
7      parse(json: string) :T {
8          return JSON.parse(json) as T;
9      }
10 }
11
12 parseJson<{ test: number }>(json: `{"test":123}`);
   new JsonParser<Record<string, number>>().parse(json: `{"test":123}`);

```

Übung 2 – Async Google Ping Tester

15 Minuten

Hier geht es um `Promise`, `async` und `await`. Wir bauen einen Ping Tester! Das Programm soll die Antwortzeit einer URL messen und einen Fehler werfen, wenn es zu lange dauert.

Modifiziert die `index.ts`

1. Implementiert eine `async` Methode `throwAfter(ms: number)`, welche nach dem gegebenen Timeout einen Error wirft
2. Implementiert `withTimeout(promise, timeout)`, welche eine Promise in einem Timeout wrapped
3. Bonus: modifiziert `pingUrl(url)`, damit es auch die Dauer einer HTTP Request zurück gibt

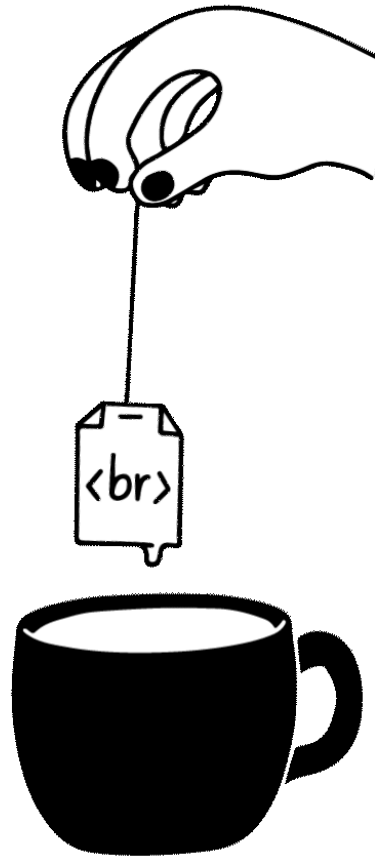
Details in der `README.md` und in den TODOs im Code



Let's go - coding time

Kurze Pause

5 Minuten



Übung 3 – Webserver

45 Minuten

Hier könnt ihr alles einsetzen, was wir gelernt haben. Der Ping Tester von vorhin geht in die nächste Runde. Dafür implementiert ihr die API für einen Webserver, wo man Pings abfragen und cachen kann. Als Bonus könnt ihr Generics Einsetzen, um JSONs effektiv auf Typen zu casten.

API:

- `/ping` für das messen einer oder mehrerer Antwortzeiten von URLs
- `/products` Bonus: eine API, welche eine andere API anfragt, den Inhalt parsed und enpackt

Details in der `README.md` und in den TODOs im Code

Cool, dass ihr dabei wart!

Übungen, Referenz-Beispiele und Folien:

<https://github.com/SoulKa/typescript-tutorial>

Weitere Learnings:

- [Async Iterators](#)
- [Streams](#)
- [Conditional Types](#)
- [Declaration Files](#) (d.ts)

Weitere Tutorials:

- <https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/>
- <https://www.typescripttutorial.net/>



Object Destructuring

Genauso, wie man Variablen in Arrays plazieren kann, kann man diese wiederum auch aus dem Array ziehen

Bei gemischten Arrays ist eine Typ Annotation hilfreich → Tupel

Der Spread Operator (`...`) extrahiert alle Werte eines Arrays. So kann ein Array genutzt werden, um alle Parameter einer Methode zu füllen.

Dasselbe geht auch mit Objekten.

```

1 // without array type annotation
2 const myArray : (string | number)[] = ["first", 1234, "third"];
3 const [a : string | number , b : string | number , c : string | number ] = myArray;
4 console.log(a, b, c);
5
6 // with array type annotation
7 const myArray2 : [string, number, string] = ["first", 1234, "third"] as [string, number, string];
8 const [d : string , e : number , f : string ] = myArray2;
9 console.log(d, e, f);
10
11 // the spread operator can be used to deconstruct one level of an array. The items are spread
12 const someNumbers : number[] = [1, 2, 3, 4, 5, 6, 7, 8, 9];
13 console.log(Math.min(...someNumbers));
14 // same as
15 console.log(Math.min.apply(Math, someNumbers));
16 // or
17 console.log(Math.min(values: 1, 2, 3, 4, 5, 6, 7, 8, 9));
18
19 // same goes for objects
20 const obj : {one: number, two: (function()... = {one: 1, two: () : number => 2, three: "3"};
21 const {one : number , two, three : string } = obj;
22 console.log(one, two(), Number.parseInt(three));

```