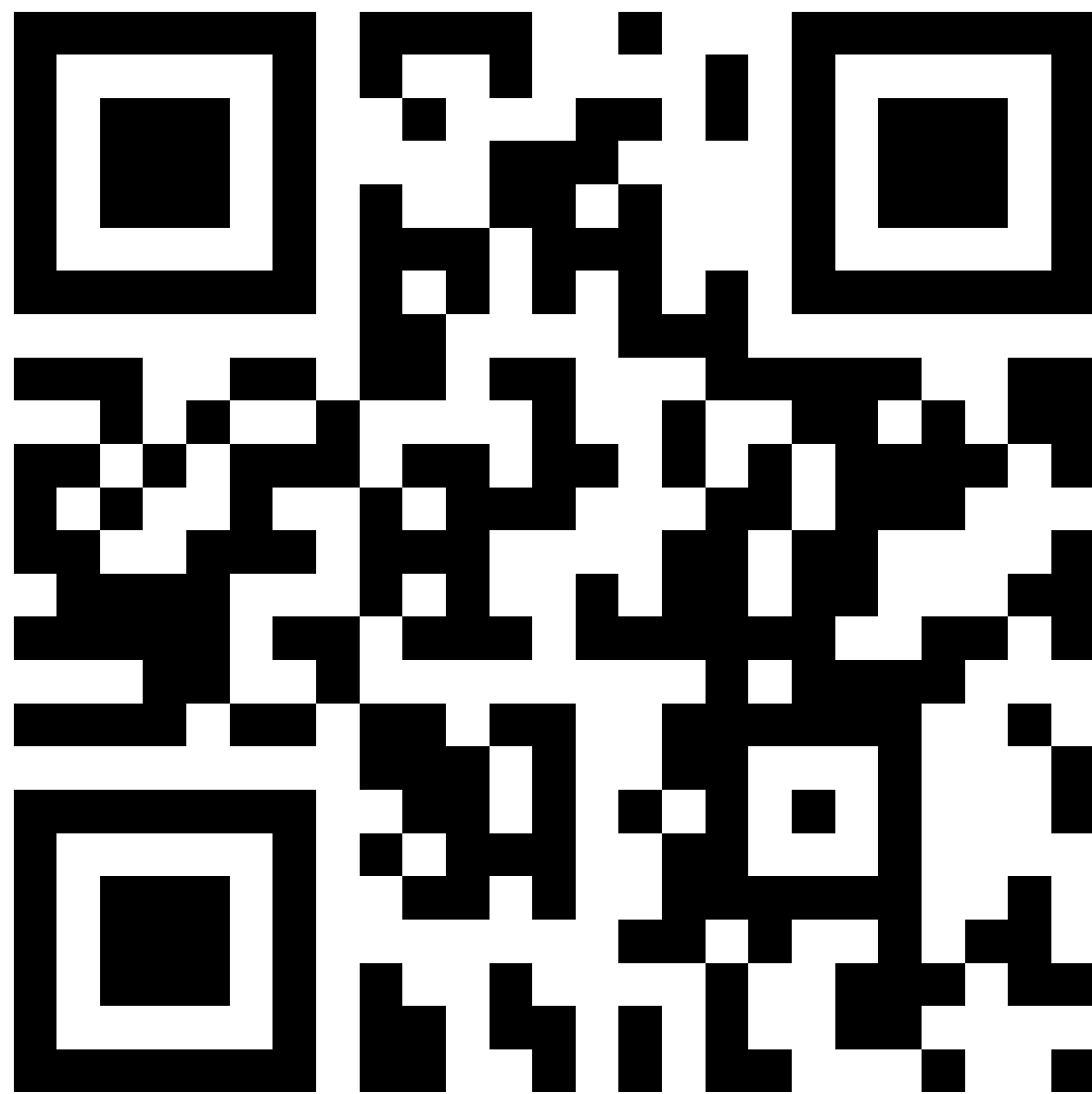


Web Engineering

DHBW Stuttgart · Informatik / Wirtschaftsinformatik

Sommersemester 2026 · Sitzung 8 — TypeScript

<https://librete.ch/dhbw/>



TypeScript

JavaScript on Steroids

Inhalt

1. TypeScript: was und warum
2. `tsc` + `ts-node`
3. `tsconfig.json`
4. Type Primitives
5. Union Types, Type Alias, Interface
6. Enum, Generics, Type Casting
7. Compilation: types stripped
8. Decorators + reflect-metadata

TypeScript – Was?

- **Superset von JavaScript** (Microsoft, 2012)
- Jede `.js` lässt sich zu `.ts` umbenennen
- **Type Errors zur Compile-/Transpile-Zeit**
- Neuere ECMAScript-Features verfügbar
- **Im Browser:** läuft nicht direkt – wird zu JS kompiliert

Setup: tsc + ts-node

```
npm i -D typescript ts-node @types/node  
npx tsc --init           # erzeugt tsconfig.json
```

Run:

```
npx ts-node index.ts      # statt: node index.js
```

Build:

```
npx tsc                   # kompiliert nach ./dist  
node dist/index.js
```

tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2022",
    "module": "commonjs",
    "moduleResolution": "node",
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Type Primitives

```
let loading: boolean = false;
let count: number = 42;
let pi: number = 3.14;
let name: string = "DHBW ist fetzig!";
let empty: null = null;
let missing: undefined = undefined;

function log(msg: string): void {
  console.log(msg);
}
```

`any`, `unknown`, `never`, `void` sind weitere TS-Spezialtypen.

Union Types

```
let id: number | string;  
  
id = 101;           // ok  
id = "101";         // ok  
id = true;          // TypeError (nicht erlaubt)
```

Type Alias

```
type NumOrString = number | string;

let id: NumOrString;
id = 101;
id = "abc";

type UserId = string;
type Status = "active" | "inactive" | "pending";
```

Interface

```
interface User {  
  id: number;  
  name: string;  
  email?: string;           // optional  
  readonly createdAt: Date; // schreibgeschützt  
  greet(msg: string): void;  
}  
  
const u: User = {  
  id: 1,  
  name: "Lisa",  
  createdAt: new Date(),  
  greet(msg) { console.log(msg); }  
};
```

Interface vs Type

Interface

- nur Objektformen
- kann mehrfach deklariert (Merging)
- bei perf-kritischen Checks oft schneller

Type Alias

- alles: Primitive, Union, Tuple, Function-Signaturen
- nicht erweiterbar nach Definition

→ Faustregel: **Interfaces für API-Shapes, Types für alles andere.**

Enum

```
enum Direction {  
    Up = 1,  
    Down,    // 2  
    Left,    // 3  
    Right    // 4  
}  
  
function walk(d: Direction): string {  
    switch (d) {  
        case Direction.Up:    return "bow";  
        case Direction.Down:  return "stern";  
        case Direction.Left:  return "windward";  
        case Direction.Right: return "leeward";  
    }  
}  
  
walk(Direction.Up); // "bow"
```

Generics

```
class GenericContainer<T> {  
    private readonly value: T;  
  
    constructor(value: T) {  
        this.value = value;  
    }  
  
    getValue(): T {  
        return this.value;  
    }  
}  
  
const numbers = new GenericContainer<number>(10);  
const strings = new GenericContainer<string>("Hello");  
  
console.log(numbers.getValue()); // 10  
console.log(strings.getValue()); // "Hello"
```

Type Casting

```
let someValue: any = "this is a string";

let len: number = (someValue as string).length;
console.log(len); // 16

// Alt-Syntax (nicht in TSX):
let len2: number = (<string>someValue).length;
```

as unknown as T für "double cast" – nur Notbehelf!

Compilation: Types werden gestripped

TypeScript:

```
type Result = "pass" | "fail";

function verify(result: Result) {
  if (result === "pass") console.log("Passed");
  else console.log("Failed");
}
```

JavaScript (nach `tsc`):

```
function verify(result) {
  if (result === "pass") console.log("Passed");
  else console.log("Failed");
}
```

Types existieren nur zur Compile-Zeit. Kein Runtime-Check.

Decorators

Meta-Programmierung mit TypeScript

TypeScript: Decorators

Vier Typen:

- **Class Decorators** – auf ganze Klasse
- **Property Decorators** – auf Felder
- **Method Decorators** – auf Methoden
- **Parameter Decorators** – auf Methoden-Parameter

Genutzt von: Angular, NestJS, TypeORM, class-validator.

Decorators aktivieren

tsconfig.json:

```
{  
  "compilerOptions": {  
    "experimentalDecorators": true,  
    "emitDecoratorMetadata": true  
  }  
}
```

Root-File:

```
import "reflect-metadata";
```

Decorator-Signaturen

```
type ClassDecorator = <T extends Function>(target: T) => T | void;

type PropertyDecorator = (
  target: Object,
  propertyKey: string | symbol
) => void;

type MethodDecorator = <T>(
  target: Object,
  propertyKey: string | symbol,
  descriptor: TypedPropertyDescriptor<T>
) => TypedPropertyDescriptor<T> | void;

type ParameterDecorator = (
  target: Object,
  propertyKey: string | symbol | undefined,
  parameterIndex: number
) => void;
```

Decorator – Beispiel

```
function Log(target: any, key: string, desc: PropertyDescriptor) {  
  const original = desc.value;  
  desc.value = function (...args: any[]) {  
    console.log(`→ ${key}(${JSON.stringify(args)})`);  
    return original.apply(this, args);  
  };  
}  
  
class Calculator {  
  @Log  
  add(a: number, b: number) {  
    return a + b;  
  }  
}  
  
new Calculator().add(2, 3); // logs: → add([2,3])
```

Reflect: Meta-Daten lesen

```
import "reflect-metadata";

Reflect.defineMetadata("role", "admin", target);
Reflect.hasMetadata("role", target);    // true
Reflect.getMetadata("role", target);    // "admin"
```

Frameworks (NestJS, TypeORM) nutzen das, um zur **Runtime** Type-Informationen zu rekonstruieren – z. B. für DI, ORM-Mapping, Validierung.

Anwendung: NestJS-Beispiel

```
@Controller("users")
export class UsersController {
  constructor(private users: UsersService) {}

  @Get(":id")
  findOne(@Param("id") id: string) {
    return this.users.find(+id);
  }
}
```

`@Controller`, `@Get`, `@Param` sind Decorators – Metadaten $\Rightarrow$ Routing.