

TypeScript

ein Superset von JavaScript

Informatik – 3. Semester

- Release 2012, entwickelt von Microsoft
- TypeScript ist ein SuperSet von JavaScript
 - Jedes JavaScript Programm ist ein valides TypeScript Programm
- TS ist ein Drop-In Replacement. Ein JavaScript Codebasis kann somit iterativ durch TypeScript ersetzt werden
- Strongly typed language

1 Motivation & Kontext

3 Syntax & Einstieg

5 Funktionen & Klassen

2 JavaScript-Recap

4 Grundlagen des Typsystems

Wie passt das zu ihrem Projekt?

- Gemeinsames Ziel: stabile Web-App im Team entwickeln
- TS ergänzt vorhandenes Wissen in HTML/CSS/JS um Typsicherheit
- Bessere Wartbarkeit, frühere Fehlermeldungen im Editor
- Erwartung: Schnellere Reviews, weniger Runtime-Bugs



- Typen für eure Domäne und die JSON-Daten der API einmal festlegen
- Erleichtert Pair Programming & Code Reviews im Team
- Auch im Back-End verbreitet (z. B. Node.js), bei uns läuft das Back-End mit Spring Boot

```
// calc-total.js
function calcTotal(price, tax) {
  return price + tax; // führt zu String-Konkatenation bei price="19"
}
```

```
// calc-total.ts
function calcTotal(price: number, tax: number) {
  return price + tax;
}
```

- Editor warnt sofort, wenn `price` als String übergeben wird

- Notiert zu zweit ein Beispiel, wo fehlende Typen euch bereits Zeit gekostet haben
- Welche Datenstrukturen waren beteiligt?
- Wie hätte eine Typannotation geholfen?

1min

1 Motivation & Kontext

3 Syntax & Einstieg

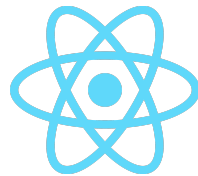
5 Funktionen & Klassen

2 **JavaScript-Recap**

4 Grundlagen des Typsystems

Schmerzpunkte an JS erkennen

- Variablen mit `var`, `let`, `const` und grundlegende Scoping-Regeln
- First-Class Functions, Arrow Functions, Promises
- DOM-Manipulation, Fetch API, Event Loop Grundverständnis



- JavaScript erlaubt spontane Typwechsel (Numbers, Strings, Objects)
- Flexibel für Prototyping, aber riskant bei Teamarbeit
- Beispiel von impliziter Typkonvertierung

```
console.log('5' - 2); // 3  
console.log('5' + 2); // "52"
```

Wer hat so etwas schon einmal im Debugger gesehen?

- Häufig: Funktionen erwarten Zahlen, bekommen aber Strings oder undefined
- Beispiel aus einer fiktiven Preisberechnung:

```
function applyDiscount(price, factor) {  
  if (!factor) {  
    return price; // vergisst, dass factor=0 gültig wäre  
  }  
  return price * factor;  
}
```

Ohne Typen/Null-Checks schwer im Review zu erkennen
Wie hättet ihr `factor` abgesichert?

- Frontend-Formulare liefern teils unvollständige Objekte
- Beispiel: API-Antwort ohne `email`

```
const user = JSON.parse(payload);  
console.log(user.email.toLowerCase()); // Runtime-Error wenn email fehlt
```

Typen könnten `email?` markieren und sofort warnen

- Wo habt ihr zuletzt Typfehler erlebt?
- Welche Daten flossen durch die Funktion?
- Warum blieb der Fehler bis in die Runtime verborgen?

```
const items = ['3', '4', '5'];  
const sum = items.reduce((acc, value) => acc + value);  
console.log(sum);
```

- Überlegen, was geloggt wird und warum
- Welche Information fehlt dem Editor, um uns zu warnen?

1 min

1 Motivation & Kontext

2 JavaScript-Recap

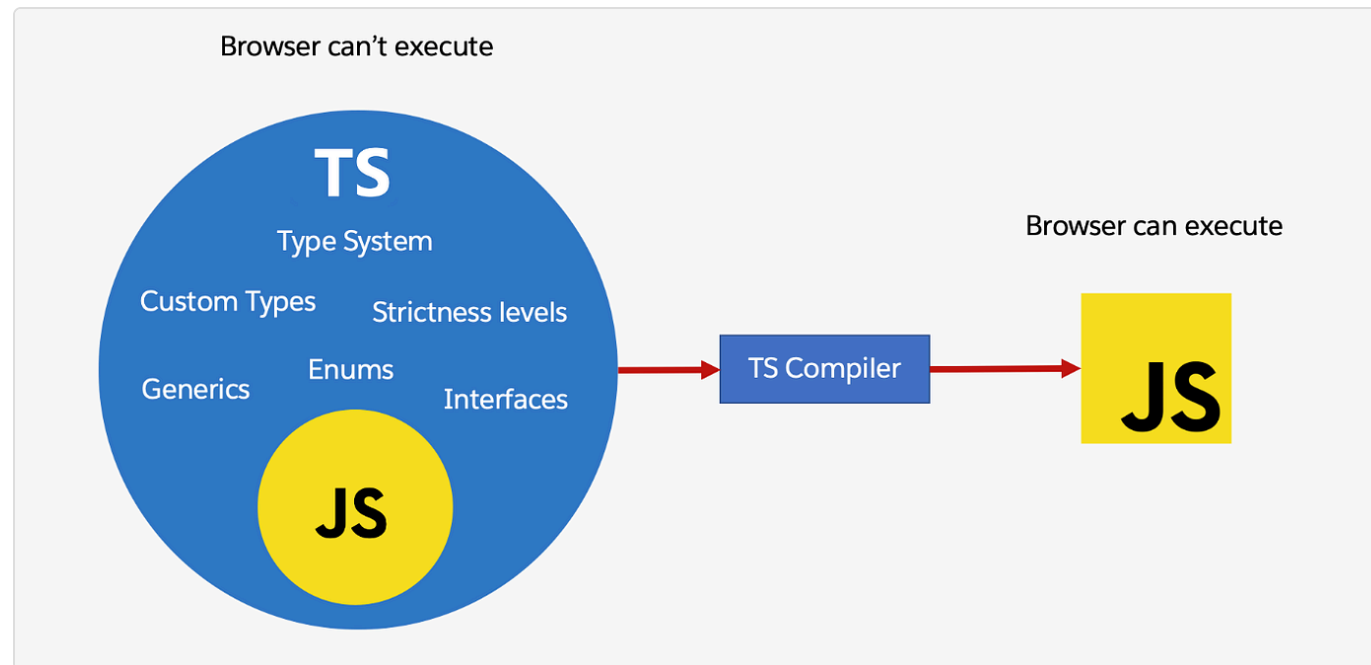
3 Syntax & Einstieg

4 Grundlagen des Typsystems

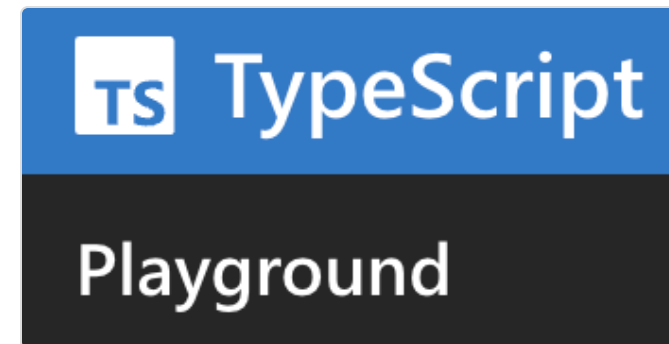
5 Funktionen & Klassen

Wie hängt TS mit JS zusammen?

- .ts-Dateien werden zu .js transpiliert
- TypeScript bietet Typprüfung, die im Browser nicht sichtbar ist
- JS-Features bleiben gültig – TS ist ein Superset



- 1. `npm init -y` – Grundgerüst anlegen
- 2. `npm install typescript --save-dev`
- 3. `npx tsc --init` – erzeugt `tsconfig.json`
- Tipp: `npx tsc --watch` für Live-Feedback



TypeScript Playground
<https://www.typescriptlang.org/play>

- `target` : Welcher JS-Standard wird erzeugt?
- `module` : CommonJS vs. ES Modules
- `strict` : Aktiviert strenge Typprüfungen
- `esModuleInterop` : erleichtert Import von CommonJS-Paketen

```
{  
  "compilerOptions": {  
    "target": "es2016",  
    "module": "commonjs",  
    "esModuleInterop": true,  
    "forceConsistentCasingInFileNames": true,  
    "strict": true,  
    "skipLibCheck": true  
  }  
}
```

- Schritt 1: Datei in `sum.ts` umbenennen
- Schritt 2: Parameter annotieren

```
// sum.js
function sum(a, b) {
  return a + b;
}

const result = sum('3', 4);
```

```
// sum.ts
function sum(a: number, b: number): number {
  return a + b;
}

const result = sum('3', 4); // Compilerfehler
```

Editor zeigt roten Unterstrich + Typ-Hinweis

- `npx tsc test.ts` erzeugt `test.js`
- Ergebnisdatei zeigt reines JavaScript, keine Typinformationen mehr
- Erinnerung: Typsicherheit existiert nur zur Compile-Zeit

```
// test.ts  
const str: string = "Hello, TS!";  
const x: number = 42;
```

```
// test.js  
var str = "Hello, TS!";  
var x = 42;
```

1 Motivation & Kontext

3 Syntax & Einstieg

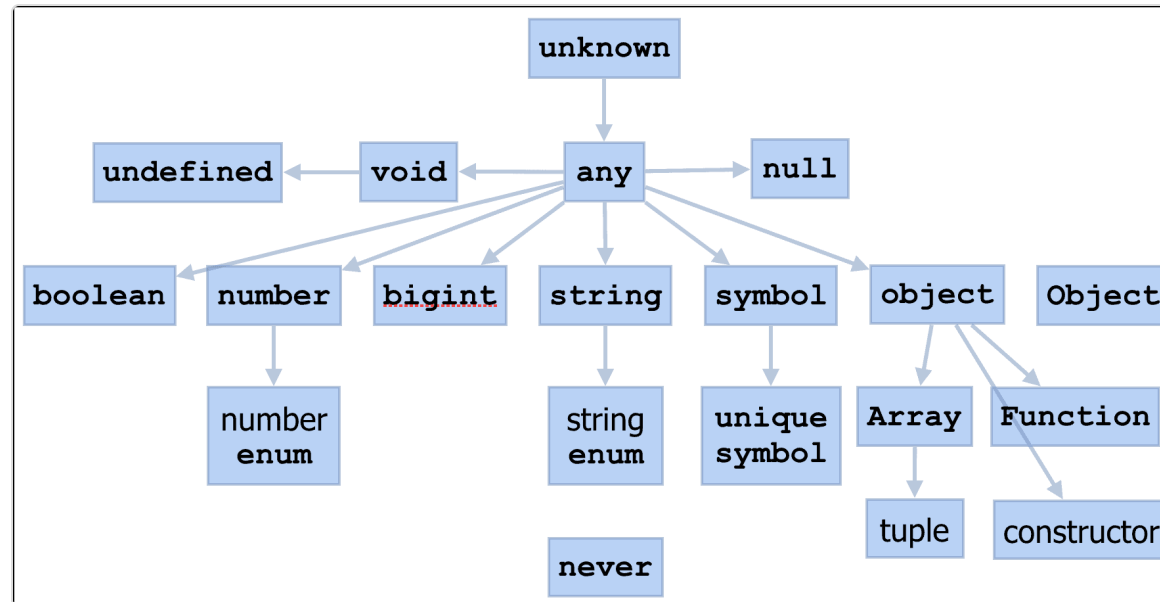
5 Funktionen & Klassen

2 JavaScript-Recap

4 Grundlagen des Typsystems

Grundlagen des Typsystems

- TypeScript erweitert JavaScript um ein statisches, strukturelles Typsystem
- Werteformen beschreiben, Vertrauen in Datenflüsse erhöhen
- Typdefinitionen existieren nur beim Entwickeln und Kompilieren



- `string`, `number`, `boolean`, `null`, `undefined`, `bigint`, `symbol`
- Literaltypen präzisieren Werte
 - `type Status = 'open' | 'done'`

```
let counter: number = 0;  
const status: 'open' = 'open';  
let flag: boolean | undefined;
```

Hinweis: `any` deaktiviert Schutz – sparsam einsetzen

- Arrays typisieren mit `string[]` oder `Array<string>`
- Tupel fixieren Länge & Positionstypen

```
const tags: string[] = ['ts', 'lecture'];  
const point: [number, number] = [12, 8];  
point[0] = 5;           // ok  
point.push(9);          // Fehler in strict mode
```

Wann würdet ihr ein Tupel statt eines Objekts nutzen?

- Union (`|`): Wert darf mehrere Formen haben
- Intersection (`&`): Kombination mehrerer Typanforderungen

```
type ID = string | number;  
type Timestamped = { createdAt: Date } & { updatedAt: Date };  
  
function printId(id: ID) {  
  console.log(id.toString());  
}
```

- Interfaces für Objektformen mit möglicher Erweiterung
- Type Aliases flexibler (Unions, Primitives, Mapped Types)
- `extends` vs. `&` zur Kombination

```
interface User {  
  id: number;  
  name: string;  
}  
  
type Admin = User & { role: 'admin' };
```

- Ziel: Interface für ein Frontend-Formular “Team-Anmeldung”
- Felder: `teamName`, `members` (Array von Strings), `contactEmail?`
- Status-Feld als Literal-Union (`'draft' | 'submitted'`)
- Optional: Tupel `[string, number]` für `preferredSlot`

3 min

```
interface TeamRegistrationForm {  
  teamName: string;  
  members: string[];  
  contactEmail?: string;  
  status: 'draft' | 'submitted';  
  preferredSlot?: [string, number];  
}
```

1 Motivation & Kontext

3 Syntax & Einstieg

5 Funktionen & Klassen

2 JavaScript-Recap

4 Grundlagen des Typsystems

... und ein Exkurs zu Generics

- Bausteine für komplexere Module:
 - Funktionen kapseln Logik
 - Klassen modellieren Strukturen
 - Generics halten Code wiederverwendbar
- Ziel: Typsicherheit ohne Doppelarbeit erreichen

- Parameter und Rückgabewerte explizit annotieren
- Optional mit `?`, Defaultwerte behalten Typen
- Rest-Parameter nutzen `...args: number[]`

```
function greet(name: string, title?: string): string {  
    return `Hallo ${title ?? ''} ${name}`.trim();  
}
```

```
const sum = (...values: number[]): number =>  
    values.reduce((acc, value) => acc + value, 0);
```

Vorteil: Editor kennt Signaturen und Autocomplete

- Function Type Aliases beschreiben Callback-Signaturen
- Überprüfbar für Events, Promises, Higher-Order Functions

```
type Comparator<T> = (a: T, b: T) => number;  
const compareUsers: Comparator<{ id: number }> = (a, b) => a.id - b.id;
```

Wie stellt ihr sicher, dass ein Callback wirklich async ist?

- Klassen deklarieren Felder, Konstruktoren und Methoden
- Modifier: `public`, `private`, `protected`, `readonly`

```
interface Notifiable {  
    notify(message: string): void;  
}  
  
class User implements Notifiable {  
    constructor(private readonly id: number, public name: string) {}  
  
    notify(message: string): void {  
        console.log(`[${this.id}] ${message}`);  
    }  
}
```

- `extends` ermöglicht Code-Sharing
- `abstract` Klassen definieren Pflichtmethoden

```
abstract class BaseService {  
    abstract execute(): Promise<void>;  
  
    log(msg: string) {  
        console.log(`[Service] ${msg}`);  
    }  
}  
  
class MailService extends BaseService {  
    async execute() {  
        this.log('Sende Mail');  
    }  
}
```

- Generics = Platzhalter für Typen (`<T>`)
- Nutzen, wenn sich Logik nicht auf einen konkreten Typ festlegen soll
- Alltägliche Beispiele: `Promise<T>` , `Array<T>` , `Map<K, V>`

```
function wrapInArray<T>(value: T): T[] {  
    return [value];  
}
```

```
const names = wrapInArray('Ada');  
const flags = wrapInArray(true);
```

- Einschränken mit `extends`, damit bestimmte Eigenschaften vorhanden sind

```
interface Identifiable {  
    id: string;  
}  
  
function findById<T extends Identifiable>(items: T[], id: string): T | undefined {  
    return items.find(item => item.id === id);  
}
```

Ohne Constraint könnte `item.id` nicht genutzt werden

```
interface Identifiable {  
    id: string;  
}  
  
class Repository<T extends Identifiable> {  
    private items = new Map<string, T>();  
  
    add(item: T) {  
        this.items.set(item.id, item);  
    }  
  
    getAll(): T[] {  
        return [...this.items.values()];  
    }  
}
```

- Welche Stellen im Projekt profitieren am meisten von Generics?
- Wo können Generics zu kompliziert werden?

- Entwerft einen generischen Typen Result
- Objekt mit generischer `data` oder `error` als String
- Definition mit Union + Generics

4 min

- Entwerft einen generischen Typen Result
- Objekt mit generischer `data` oder `error` als String
- Definition mit Union + Generics

```
type Result<T> = { data: T } | { error: string };
```

4 min