



thecodecampus</>

Alle ♥ TypeScript - und du?



TypeScript

Philipp Burgmer

<> Software-Entwickler, Trainer
Web-Technologien
burgmer@w11k.de </>

w11k GmbH - The Web Engineers

<> Individual Software für Geschäftsprozesse
Entwicklung, Wartung & Betrieb
Consulting </>

TheCodeCampus - Weiter.Entwickeln.

<> Technologie Schulungen
Projekt Kickoff, Code Reviews
Angular, TypeScript </>

Warum TypeScript

Was sind die Probleme mit JavaScript

Warum TypeScript

<> Besseres, zuverlässigeres Tooling

- Fehler vermeiden / früher finden
- Wie sieht die API noch mal aus?
- Wo ist das noch mal definiert?

<> Lösung bisher

- Linting
- IDEs, die raten
- Disziplin im Team bei Namensgebung und Struktur

Warum TypeScript

<> Große Projekte

- in ES5 schwierig zu entwickeln und warten
- Große Teams
- Große Code-Basis
- Lange Wartungszeiträume

<> Lösung bisher

- Viel Dokumentation
- Viele Tests

<> Viel Aufwand

- ES6 hilft an vielen Stellen enorm weiter
- Klassen, Module, Arrow-Functions, ...

Warum TypeScript

<> JavaScript von Morgen schon heute

- ES6+ in Browser teils nicht verfügbar
- Browser hinken immer hinterher

<> Lösung bisher

- Polyfills
- Transpiler (z.B. Babel)

Wie hilft uns TypeScript

Was ist TypeScript eigentlich

<> Keine eigene Sprache

- Superset von EcmaScript
- Vereinfacht: EcmaScript erweitert um Typisierung
- Wird nicht ausgeführt, kein Interpreter
- Wird zu ES Code übersetzt

<> Entwickelt von Microsoft

- Apache 2 Lizenz, Open Source
- Cross Platform, Cross IDE

<> Wenn `TypeScript = ES + Typisierung`

<> Dann `TypeScript - Typisierung = ES`

<> **TypeScript Compiler = ES Transpiler**

- TypeScript Compiler gibt ES3/5/6 Code aus
- ES6 Features heute schon nutzen

<> Leichter Einstieg möglich

- Gleiche Syntax
- Alle Features optional
- Klassen, Module, Typen, ...

<> TypeScript Compiler vorausgesetzt

- Muss in Tooling-Pipeline integriert werden

```
npm install typescript -g  
tsc hello-world.ts  
node hello-world.js
```

- <> Angular 2 komplett in TypeScript entwickelt
 - Klassen, Interface
 - Modul-System
 - Decorators
- <> Enge Zusammenarbeit der Teams bei Google und Microsoft
- <> Große Code-Basis -> profitiert stark von Struktur-Features

<> Kein Zwang TS zu nutzen

- ES Code + Type Declarations ausgeliefert
- Voraussetzung: ES5 + Module-Loader

<> Aber Empfehlung

<> Typen

<> ES6+

<> Klassen & Interfaces

<> Generics

<> Module

<> Decorators

<> Tooling

<> Bonus

- Null- and undefined-aware types
- async/await

Typen

<> Wann wird der Typ geprüft?

- **statische** Typisierung: `String s = new String();`
- **dynamische** Typisierung: `var s = "a"; s = 3;`

<> Wie streng sind die Typen?

- Wie viel wird automatisch konvertiert?
- **schwache** Typisierung: `s = "Zahl " + 5;`
- **starke** Typisierung: `s = "Zahl " + str(5)`

<> JavaScript: dynamisch, sehr schwach

<> TypeScript: statisch, etwas stärker aber immer noch schwach

<> Fehler früher finden

<> Besseres Tooling, IDE weiß mehr über den Code

- Sinnvolle Autovervollständigung
- Vorschau auf Signatur
- Refactoring
- Navigation
- Find References

Typen - Warum lieber nicht

- <> Statische Typ-System stehen oft im Weg
- <> EcmaScript ist so schön flexibel
- <> TypeScript verbindet beides

My favorite [TypeScript] feature is that the type system mimics the actual JS spec as well as the common JS practices in the community very closely, and so it feels very natural to use.

Misko Hevery, Angular Team

<> Basis-Typen: `string`, `number`, `boolean`

<> Typ für Array

<> Eigene Typen über Klassen, Interfaces, Enums, Funktionen

<> Union-Types

<> String-Literal-Types

<> ...

<> Variablen

- JavaScript: `var x = "Hello";`
- TypeScript: `var x: string = "Hello";`

<> Funktionen

- JavaScript: `function sum(a, b) {}`
- TypeScript: `function sum(a: number, b: number): number {}`

<> Typen können *explizit* angegeben werden

```
1 var x: string = "Hello";  
2 name = 5; // COMPILE ERROR
```

<> Oder *inferred* werden

```
1 var x = "Hello"; // 'x' is of type string  
2 name = 5; // COMPILE ERROR
```

<> **any** Type

```
1 var x: any = "Hello";  
2 name = 5; // OK  
  
1 // 'a' and 'b' are of type any  
2 function (a, b: any) {}
```

Demo

Typen

ES6+

Das Problem mit var

<> Welche Werte hat `a` und `b`?

```
1  var a = 1;
2  var b = 2;
3
4  if (true) {
5      var a = 98;
6      b = 99;
7  }
8
9  console.log(a);
10 console.log(b);
```

<> Variablen mit Schlüsselwort `var` sind in EcmaScript *function scoped*

<> Schlüsselwort `let` statt `var` sorgt für *block scope*

```
1  let a = 1;
2  let b = 2;
3
4  if (true) {
5      let a = 98;
6      b = 99;
7  }
8
9  console.log(a);
10 console.log(b);
```

<> Schlüsselwort `const` markiert Variablen als unveränderbar

<> *block scoped* wie bei `let`

```
1  const x = { a: 1 };
2  x = { a: 2 }; // COMPILER ERROR
3  x.a = 2 // ok
4
5  console.log(x);
```

this innerhalb von Funktionen

<> Gegeben sei folgender Code:

```
1 function Person() {  
2   this.age = 0;  
3  
4   setInterval(function () {  
5     this.age++;  
6   }, 1000); // every second  
7 }
```

<> Wie ist die Ausgabe bei folgendem Code?

```
1 var p = new Person();  
2 setInterval(function () {  
3   console.log(p.age);  
4 }, 10000); // every 10 seconds
```

<> Kürzere Syntax

- `(a, b) => {...}` statt `function (a, b) {...}`
- Klammern `()` sind bei nur einem Parameter optional
- Klammern `{}` und `return` sind optional, wenn *body* nur aus einem Statement besteht

```
1 let add = (a, b) => a + b;  
2 console.log(add(1, 2)); // 3
```

<> `this`-Capturing

- Einsatzgebiet: Callbacks, Promise-Handler, ...

```
1 function Person() {  
2   this.age = 0;  
3   setInterval(() => { this.age++; }, 1000); // every second  
4 }
```

<> for..in Schleife in JavaScript ist umständlich und fehleranfällig

```
1 let list = [1, 2, 3];
2 list.something = "ups";
3
4 for (let i in list) {
5     console.log(list[i]);
6 }
7 // output: 1, 2, 3, ups
8 // maybe: 2, ups, 1, 3
```

<> for..of verhält sich wie erwartet

```
1 for (let i of list) {  
2   console.log(i);  
3 }  
4 // output: 1, 2, 3
```

- Iteriert über Iterable (Array, Map, Set, NodeList, ...)
- Garantiert Reihenfolge
- Value statt Key in Schleifenvariable `i`

<> Einsatzgebiet: Variable Parameterlisten

- Statt `arguments` Variable

<> Argumente als echtes Array entgegennehmen

- Impliziter Type `any[]`
- Explizit angeben z.B. `number[]`

```
1 function fn(a, b, ...derRest) {  
2   console.log(derRest);  
3 }  
4  
5 fn(1, 2); // []  
6 fn(1, 2, 3); // [3]  
7 fn(1, 2, 3, 4); // [3, 4]
```


<> Kein Überladen von Funktionen / Methoden in EcmaScript

<> Parameter sind daher oft optional

- Mit `?` als *optional* kennzeichnen (Wert: `undefined`)
- Mit `=` einen Standard-Wert zuweisen

```
1 function withOptionalParams(regular, optional?, defaultValue = 1) {  
2     console.log(regular, optional, defaultValue);  
3 }  
4  
5 withOptionalParams(0);
```

<> Syntax: Öffnen und Schließen per ``` (Backtick)

```
1 let text: string = `Hallo Welt!`;
```

<> Multiline Strings

```
1 let text = `Dies ist ein String  
2 der einen Zeilenumbruch enthält.`;
```

<> String Interpolation

```
1 let name = "Max";  
2 let text = `Hallo ${name}!`;
```

Klassen & Interfaces

<> Mit *Interfaces* können Strukturen Namen gegeben werden

<> Reines TypeScript Feature

- Dienen nur der Typ-Prüfung
- Erzeugen keinen ECMAScript Code im Output
- Werden für gewöhnlich nur an System-Grenzen verwendet

```
1 interface HasLength {  
2   length: number;  
3 }
```

<> Interfaces können Variablen und Methoden definieren

<> Methoden immer ohne Implementierung

```
1 interface HasAge {  
2   birthDate: Date;  
3   getAge(): number;  
4 }
```

- <> Typ-System von TypeScript ist *strukturell*
- <> a.k.a. duck-typing
- <> Java & Co. sind *nominal*
- <> Typen sind kompatibel, wenn deren Struktur gleich ist
- <> Name / Erzeuger spielt keine Rolle

```
1 interface HasLength { length: number }
2
3 function printLength(obj: HasLength) {
4     console.log("Länge: " + obj.length);
5 }
6
7 let o = { length: 10 };
8 printLength(o);
9 printLength([1, 2, 3]);
10 printLength("Hello TypeScript");
```

- <> Klassen kombinieren Schnittstelle und deren Implementierung
- <> Zugriff untereinander auf *Member-Variablen* mittels `this`
- <> KEIN automatisches `this`-Capturing für Methoden
 - Methoden hängen am Prototype
 - Ansonsten müssten es Instanz-Variablen sein (schlechtere Performance)
- <> Klassen ES6-Feature, Typisierung über Klassen TS-Feature

```
1 class MyClass {  
2   aMember: number = 123;  
3   aMethod(param: string): string {  
4     console.log(this.aMember); // 123  
5     return param.toUpperCase();  
6   }  
7 }  
8 let anInstance = new MyClass();
```


<1> Erstelle die Klasse `Student`,
so dass folgender Code kompiliert und ausgeführt werden kann

```
1 let s: Student = new Student();  
2 let label: string = s.getLabel(123456);  
3 console.log(label); // "Student mit Matrikelnummer: 123456"
```

- <> Ermöglicht Werte bei der Erzeugung des Objektes mitzugeben
- <> Sinnvoll für *Pflichtfelder*
- <> Wird als Methode mit dem Namen `constructor` definiert

```
1 class MyClass {  
2   constructor(wert: string) {  
3     // wert benutzen  
4   }  
5 }  
6 let myc: MyClass = new MyClass("ein Wert");
```

<1> Sorge dafür, dass folgenden Code kompiliert und ausgeführt werden kann

```
1 let s: Student = new Student("Max", 123456);  
2 let label: string = s.getLabel();  
3 console.log(label); // "Student Max mit Matrikelnummer: 123456"
```

<> Zugriff von "außen" auf Klassen-Eigenschaften kann eingeschränkt werden

<> `public`, `private` und `protected`

<> Default: `public`

<> Reines TypeScript Feature, in JS immer alles `public`

```
1 class {  
2   private aMember = 123; // default: public  
3  
4   private aMethod(): {}  
5 }
```

<> Gängiger Code

```
1 class Student {  
2   _name: string;  
3   constructor(name: string, mnr: number) {  
4     this._name = name;  
5   }  
6 }
```

<> Instanzvariablen per Konstruktor-Parameter direkt in Parameterliste anlegen

```
1 class MyClass {  
2   constructor(public wertA: string, private wertB: number) {}  
3 }
```

<> Interfaces und Klassen können sinnvoll kombiniert werden

<> Interfaces definieren Schnittstellen und fördern die Wiederverwendbarkeit

```
1 interface Labelled {  
2     getLabel(): string;  
3 }
```

<> Klassen und Objekte werden per `structural typing` mit Interfaces verglichen

```
1 let withLabel: Labelled = new Student("Max", 123456);
```

<> Klassen können Interfaces *explizit implementieren*

```
1 class Student implements Labelled {}
```

- Schon bei Erstellung sichergestellt, dass Klasse *strukturell* zu Interface kompatibel

<> Ermöglicht es, sich wiederholende Strukturen wiederzuverwenden

<> Klassen können ihre Eigenschaften vererben

- Expliziter Zugriff auf Member der Basisklasse über `super`
- Super-Konstruktor muss explizit aufgerufen werden `super(arg1, arg2, ...)`
- Kein Überschreiben von Properties (Instanz), nur Methoden (Prototyp)

```
1 class Student extends CampusPerson {  
2   constructor() { super(1); }  
3   protected aMethod() { super.aMethod(); }  
4 }
```

- <> Schlüsselwort `abstract` an Klasse verhindert Instanziierung
- <> Methoden können mit `abstract` markiert werden -> ererbte Klassen muss diese implementieren oder ebenfalls `abstract` sein
- <> Unterschied zu Interfaces: erzeugen ES Code

```
1 abstract class CampusPerson {  
2   abstract morgensAmCampus();  
3 }  
4  
5 let cp = new CampusPerson(); // COMPILE ERROR
```


<> Welchen Type hat `obj` ?

```
1 let obj = {  
2   a: 1,  
3   b: "Beh"  
4 };
```

<> Object Literale führen zu *object types*

- Aus Objekt abgeleiteter Typ

<> Union Types sind die Vereinigung von 2 Typen

- Werden in der Typ Signatur über `|` verbunden
- Zuweisbar: beide Typen
- Aufrufbar: was in beiden Typen gleich definiert ist

```
1 let campusPerson: Student | Professor;  
2  
3 campusPerson = new Student("Max", 123456); // OK  
4 campusPerson = new Professor("Hugo"); // OK  
5 campusPerson = "Paul"; // Fehler
```

<> Wichtig für JavaScript-Welt

- Kein Überladen von Methoden
- Eine Methode nimmt A oder B entgegen (z.B. `string | string[]`)
- Innerhalb der Methode Typ-Prüfung (`Array.isArray`)

Generics

- <> Aus anderen Sprachen bekanntes Konstrukt
- <> Wiederverwendbarkeit + Typsicherheit
- <> Datenstrukturen wie Array von Typ der Daten unabhängig machen
 - Operationen an Struktur hängen nicht von Typ der Daten ab
`Array#pop` unabhängig von `string` oder `number` (Implementierung)
 - Trotzdem wissen was in Datenstruktur steckt
Was liefert `myArray.pop()` (Verwendung)

<> Typ-Parameter an Klasse/Interface und Methode

```
1 interface Array<T> {  
2     reverse(): T[];  
3  
4     sort(compareFn?: (a: T, b: T) => number): T[];  
5  
6     map<U>(  
7         func: (value: T, index?: number, array?: T[]) => U  
8     ): U[];  
9 }
```

<> *Type Constraints* werden unterstützt

```
1 interface MyRestrictedCollection<T extends Something> {}
```

Demo

Generics & Type-Inferencing

Module

- <> Keine Struktur in reinen ES5 Projekten
- <> Große Code-Basis == schwierige Orientierung / Navigation
- <> Zu wenig Kapselung / zu viel globales

- <> EcmaScript 5 hat kein Modul System
- <> Alle `.js`-Dateien sind Skripte, die nacheinander ausgeführt werden
- <> Probleme
 - Keine Kapselung, ein Namensraum
 - Schlechte Wiederverwendbarkeit
 - Abhängigkeiten sind nicht erkennbar

<> Lösung bisher: CommonJS (Node) oder AMD (RequireJS)

<> Probleme: Wer unterstützt was? Was liegt wie vor?

<> Lösung: ES6 Module

- Einheitliches, natives Modul System
- Datei == Modul
- Datei wird erst durch mindestens ein Import- oder Export-Statement zu Modul

<> Unterscheiden zwischen

- Syntax zum definieren und laden
- Laden und Einbinden zur Laufzeit

<> Im TypeScript Code

- ES6 Modul Syntax mit import und export Statements

<> Compiler übersetzt Syntax in API Aufrufe eines Module Loaders

- ES Module Loader API (System.import)
- AMD (RequireJS)
- CommonJS (Node)
- Native ES6 Module Syntax

Named Export

<> Top-Level Variablen können exportiert werden

<> Stehen unter ihrem Namen zur Verfügung

```
1 // my-module.ts
2 let x = 2; // not visible outside
3
4 export let aNumber = 1;
5 export let arrowFun = () => { };
6 export function fatFun() {
7     // ...
8 };
9
10 export class MyClass {
11     // ...
12 }
```

Named Import

<> **named exports** importieren

```
1 import {fatFun, MyClass} from "./my-module";
```

- <1> Lagere die Klasse `Student` in ein Module aus
 - Erstelle eine Datei `student.ts`
 - Verschiebe die Klasse dort hin und exportiere sie
- <2> Importiere die Klasse in der ursprünglichen Datei

Export- & Import Statements

- <> Alle `import` Statements sind *hoisted*
- <> `import / export` Statements dürfen nicht ...
 - variabel sein
 - conditional sein (`if`, `case`, etc.)
- <> Erlaubt statische Analyse
 - Bundling
 - Code Navigation in IDE

<> Module sollten wenn möglich relativ geladen werden (eigener Code)

```
1 import {arrowFun} from "./my-module";
```

<> Module können auch absolut geladen werden (Top-level Module)

```
1 import {_} from "lodash";
```

- Externe Bibliotheken
- Integration legacy Code
- Nicht spezifiziert von wo geladen wird
 - muss konfiguriert werden
 - für gewöhnlich `node_modules`

<> Statt import + export Statement kombinierte Schreibweise

```
1 export {fatFun, MyClass} from "./my-structure/my-module";
```

<> Bietet Möglichkeit **ein** Modul nach außen anzubieten

- Interne Struktur von Sub-Modulen verstecken
- Meist über `index.ts`
- Gut für Bibliotheken

Decorators

Decorators - Warum

- <> Framework spezifische Metadaten
- <> Möglichst Framework unabhängiger Code

- <> Ähnlich Annotations in Java und Decorators in Python
- <> Transpiler übersetzt Decorator-Verwendung in ES Code
 - Typ-Informationen aus TS Code zur Laufzeit nutzen
- <> Experimental Feature, ES Proposal

Decorators

<> Können verwendet werden an

- Klassen (Konstruktor)
- Properties & Methoden
- Parametern

<> Müssen immer (auch ohne Parameter) aufrufende Klammern haben ()

```
1 @Freezed()  
2 class Person {  
3  
4     constructor(@NotUndefined() private name:string) {}  
5  
6     @LogCall("warn")  
7     greet() { console.log(`Hello, I'm ${this.name}`); }  
8 }
```

Demo

Decorators

Tooling

- <> Editor / IDE rät bei ES Code zu viel, keine verlässlichen Informationen
- <> ES Unterstützung je nach Editor sehr unterschiedlich
- <> Entwickler sollten möglichst nicht an einen Editor gebunden sein
- <> Gute Unterstützung in möglichst vielen Editoren

<> Informationen aus Typen und Modulen

- Code zu ES übersetzen reicht nicht aus
- Typ-Informationen existieren in ES Code nicht mehr
- Tooling muss TS Code verstehen

<> Tooling nicht für jeden Editor implementieren

- TSServer läuft im Hintergrund, bietet API für Tooling
- Visual Studio, VS Code, JetBrains IDEs, Atom, Eclipse, Emacs, ...

<> Compiler Konfiguration

- In `tsconfig.json` ausgelagert
- Verschiedene Tools können Compiler anstoßen
- Immer gleiche Konfiguration

```
1 {  
2   "compilerOptions": {  
3     "target": "es5",  
4     "module": "system",  
5     "noImplicitAny": false,  
6     "rootDir": "./src",  
7     "outDir": "./target/tsc"  
8   }  
9 }  
10
```

Strict-Null

Null- and undefined-aware types

<> `null`: The Billion Dollar Mistake

<> **Compiler Flag** `--strictNullChecks`

- Typen `undefined` und `null`
- Andere Typen beinhalten Werte `null` und `undefined` nicht mehr

```
1 let x: number = undefined // error
```

- Union Types

```
1 let x: number | undefined = undefined // ok
```

Null- and undefined-aware types

<> Ermöglicht zusätzliche Checks

```
1 let x: number;
2 x; // Error, reference not preceded by assignment
3 x = 1;
4 x; // Ok
5
6 let z: number | undefined;
7 z; // Ok
```

Null- and undefined-aware types

<> Was gibt `document.querySelector("foo")` zurück?

<> Auf `null` prüfen?

<> Auf `undefined` prüfen?

<> Auf beides prüfen?

Null- and undefined-aware types

```
1 var element = document.querySelector("foo");  
2  
3 element.classList.add("bar"); // error: element can be null  
4  
5 if (element !== null) {  
6     element.classList.add("bar"); // ok  
7 }
```

Async-Await

<> Asynchrone Programmierung ist schwierig

- Stark verschachtelter Code
- Oft schlecht lesbar

<> Async/Await macht Code wieder flach!

- ES Proposal
- Leider nicht in ES2016, hoffentlich 2017
- TypeScript unterstützte schon für `target: "es6"`
- Ab 2.1 auch für ES3 und ES5

```
1  async function doSomething() {  
2    await doSomethingAsync();  
3    console.log("finished");  
4  }
```

```
1 function check() {  
2   return new Promise<boolean>(resolve => resolve(false));  
3 }  
4  
5 function doNotGiveIn() {  
6   return new Promise<void>(resolve => { console.log("Do it now!"); resolve(); });  
7 }  
8  
9 async function askForCleanup() {  
10  console.log("Please clean up your room");  
11  while(await !check()) { await doNotGiveIn(); }  
12  console.log("Thanks!");  
13 }  
14  
15 askForCleanup();
```

Zusammenfassung

- <> Leichter Einstieg möglich, sehr nah an EcmaScript
- <> ES6 + Typen
 - Mehrwert in täglicher Arbeit
 - Großer Mehrwert bei größeren Teams / Projekten / Zeitabständen
- <> Module System in TS einfach, zur Laufzeit schwierig

<> Für Angular 2+ quasi gesetzt

- Module-System von Angular verwendet
- Typ-Informationen gleich dabei
- Gute Unterstützung durch Angular-CLI

<> Gut in AngularJS 1.x integrierbar

- Herausforderung: Build-System
- Typ-Informationen verfügbar (@types)
- Mehrwert: eigenen Code besser strukturieren

Philipp Burgmer
burgmer@w11k.de
Twitter: @philippburgmer
GitHub: pburgmer

www.thecodecampus.de
@theCodeCampus