



webinale

ng2 ❤ TypeScript - und du?

Philipp Burgmer



Über mich

Philipp Burgmer

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

Über uns

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, Compiler kann einiges besser

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

Warum TypeScript

- <> JavaScript von Morgen schon heute
oder nur kein JavaScript von gestern mehr
 - ES6 in Browser nicht verfügbar
 - Browser hinken immer hinterher

- <> Lösung bisher
 - Polyfills
 - Transpiler

Wie hilft uns TypeScript

Was ist TypeScript eigentlich

<> Keine eigene Sprache

- Superset von EcmaScript
- Vereinfacht: ES6 + Typisierung
- Kein Interpreter
- TS Code wird von Compiler in ES5 Code übersetzt
- Interoperabel mit ES Code

<> Entwickelt von Microsoft

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

Transpiler

<> Wenn

- TypeScript = ES6 + Typisierung
- TypeScript - Typisierung = ES6
- TypeScript Compiler gibt ES5 Code aus

<> Dann

- TypeScript Compiler = ES6 Transpiler
- ES6 Features heute schon nutzen
- Polyfills benötigt (core-js oder es6-shim)

Einstiegshürde

<> 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
```

EcmaScript zu TypeScript portieren

<> *.js Datei in *.ts umbenennen

<> TypeScript ist Superset -> prinzipiell jeder ES Code gültig

<> Compiler kennt APIs der Umgebung (Browser mit DOM oder Node)

<> Compiler analysiert eigenen Code (Type Inferencing)

```
1 Object.defineProperty(window, "myAppVersion", {  
2   value: 'v0.1.0', readOnly: true  
3 });
```

EcmaScript zu TypeScript portieren

<> Vorsicht: Nicht jeder Fehler ist ein Fehler

```
1 let x = { a: 1 };
2 x.b = 2;
```

<> Compiler macht *Type Inferencing*

```
1 let z = "Hello World"; // z: string
```

<> Typ `any` macht alles möglich

Angular ❤ TypeScript

<> 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

Angular ❤ TypeScript

<> Kein Zwang TS zu nutzen

- ES Code + Type Declarations ausgeliefert
- Voraussetzung: ES5 + Module-Loader (z.B. SystemJS)

<> Aber Empfehlung

Features

<> Typen

<> Generics

<> Module

<> Tooling

<> Decorators

Typen

Typen - Warum

<> Fehler früher finden

<> Besseres Tooling, IDE weis mehr über den Code

- Sinnvolle Auto vervollständigung
- 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

Typen - Warum doch

My favorite [TypeScript] features 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

Typen - Wie

- <> Basis-Typen: `string`, `number`, `boolean`
- <> Typ für Array
- <> Eigene Typen über Klassen, Interfaces, Enums, Funktionen
- <> Union-Types
- <> String-Literal-Types
- <> ...

Typen - Wie

- <> Typ-System ist strukturell, nicht nominal
- <> a.k.a Duck-Typing
- <> Typ-System steht nicht im Weg

Demo

Typen

Generics

Generics - Warum

<> 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)

Generics - Wie

<> Typ-Parameter an

- Klassen und Interfaces
- Methoden / Funktionen

<> Ko- und Kontra-Varianz werden nicht unterstützt

```
1 interface Array<T> {
2     reverse(): T[];
3
4     map<U>(fun: (value: T) => U): U[];
5 }
```

Demo

Generics

Module

Module - Warum

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

Module - Wie

- <> Lösung bisher: CommonJS (Node) oder AMD (RequireJS)
- <> Probleme: Wer unterstützt was? Was liegt wie vor?
- <> Lösung: ES6 Module
 - Einheitliches, natives Modul System
- <> Unterscheiden zwischen
 - Syntax zum definieren und laden
 - Laden und Einbinden zur Laufzeit

Module - Wie

<> 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

Demo

Module

Tooling

Tooling - Warum

- <> 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

Tooling - Wie

<> 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, ...

Tooling - Wie

<> Compiler Konfiguration

- In `tsconfig.json` ausgelager
- 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 }
```

Decorators

Decorators - Warum

<> Framework spezifische Metadaten

<> Möglichst Framework unabhängiger Code

<> Aus ng1 bekannte Probleme vermeiden

- mit DI über Strings / Parameter Namen
- Komponenten an Framework bekannt machen (ng1 Module)

Decorators - Wie

- <> Ähnlich Annotations in Java und Decorators in Python
- <> Transpiler übersetzt Dectorator-Verwendung in ES Code
 - Typ-Informationen aus TS Code zur Laufzeit nutzen
 - Abfragen aller dekorierten Klassen
- <> Experimental Feature, ES7 Proposal

Demo

Decorators

Zusammenfassung

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

Philipp Burgmer
burgmer@w11k.de
@philippburgmer

www.thecodecampus.de
@theCodeCampus

