ANGULAR DAYS 2017

TypeScript

Basics

thecodecampus</>

Über mich

Philipp Burgmer
<> Software-Entwickler, Trainer
Web-Technologien
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**
- 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

Transpiler

<> **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

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

<> **Aber Empfehlung**

Einstiegshürde

<> **Leichter Einstieg möglich**
- Gleiche Syntax
- Alle Features optional
- Klassen, Module, Typen, ...

<> **TypeScript Code muss übersetzt werden**
- Compiler muss in Tooling-Pipeline integriert werden

```
npm init
npm install typescript --save-dev
./node_modules/.bin/tsc hello-world.ts
node hello-world.js
```

Hands-on: Hello World!

Hands-on

<1> Öffne ein Terminal-Fenster

<2> Erstelle irgendwo einen Ordner `hello-typescript` und navigiere hinein

<3> Führe `npm init -y` aus

<4> Führe `npm install typescript --save-dev` aus

<5> Führe `./node_modules/.bin/tsc --init` aus

<6> Führe `./node_modules/.bin/tsc --watch` aus

<7> Öffne den Ordner in der IDE

<8> Lege die Datei `src/hello.ts` mit folgendem Inhalt an:

```
1 console.log("Hello TypeScript!");
```

<9> Öffne ein weiteres Terminal und führe in `hello-typescript` `node src/hello.js` aus

Features

<> Typen

<> ES6+

<> Klassen & Interfaces

<> Generics

<> Type System Advanced

<> Module

<> Decorators

<> Bonus
- null- and undefined-aware types
- async / await

Typen

Typisierung

<> Wann wird der Typ geprüft?
- **statische** Typisierung: `String s = "foo";`
- **dynamische** Typisierung: `var s = "foo"; s = 3;`

<> Wir streng sind die Typen?
- Wie viel wird automatisch konvertiert?
- **schwache** Typisierung: `s = "foo" + 3;`
- **starke** Typisierung: `s = "foo" + str(3);`

<> JavaScript: dynamisch, sehr schwach

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

Typen - Warum

<> **Fehler früher finden**

<> **Besseres Tooling, IDE weis 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

Typen - Warum doch

> 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*

Typen - Wie

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

<> Typ für Array

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

<> Union-Types

<> String-Literal-Types

<> ...

Type Annotations

TypeScript

<> Variablen

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

<> Funktionen

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

Type Inferencing

<> Typen können *explizit* angegeben werden

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

<> Oder *inferred* werden

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

Der Typ any

<> **Fallback zu ES Verhalten über *any***

- Alles kann zugewiesen werden
- Variable kann für alles eingesetzt werden

```
1 var x: any = "Hello";
2 x = 5; // OK
3 var foo: boolean = x; // OK
```

Type Inferencing bei Funktionsparametern

<> **Für alleinstehende Funktionen nicht möglich**

- Parameter ohne expliziten Typ sind `any`
- Kann per Konfiguration verboten werden (*noImplicitAny*)

```
1 // 'a' and 'b' are both of type any: a explicit, b implicit
2 function foo(a: any, b) {}
```

<> **Type-Inferencing bei Lamdas möglich**

```
1 // type of result defined by signature of bar
2 bar(function callback(result) {});
```

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);
```

Block Scoping mit Let

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

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

```
let a = 1;
let b = 2;

if (true) {
  let a = 98;
  b = 99;
}

console.log(a);
console.log(b);
```

Konstanten mit Const

ECMAScript 6

<> Schlüsselwort `const` markiert Variablen als unveränderbar

<> *block scoped* wie bei `let`

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

this innerhalb von Funktionen

<> Gegeben sei folgender Code:

```
function Person() {
  this.age = 0;

  setInterval(function () {
    this.age++;
  }, 1000); // every second
}
```

<> Wie ist die Ausgabe bei folgendem Code?

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

Arrow Function

<> Kürzere Syntax

- `(a, b) => {...}` statt `function (a, b) {...}`
- `()`, `{}` und `return` ggf. optional

```
let powerOfTwo = a => 2 ** a; // (a: number) => { return 2 ** a; }
console.log(powerOfTwo(3)); // 8
```

<> `this`-Capturing

- `this` des Definitionskontext wird übernommen
- Aufrufer hat keine Kontrolle über `this` innerhalb der Funktion
- Einsatzgebiet: Lamdas (Callbacks, Handler, ...)

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

for .. in Schleife

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

```
let list = [1, 2, 3];
list.something = "ups";

for (let i in list) {
    console.log(list[i]);
}
// output: 1, 2, 3, ups
// maybe: 2, ups, 1, 3
```

for .. of Schleife

<> for .. of verhält sich wie erwartet

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

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

Optional Parameters

<> 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);
```

Rest Parameters

<> **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]
```

Template Strings

ECMAScript 6

<> 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

Interfaces

<> Mit *Interfaces* können Strukturen Namen gegeben werden

<> Reines TypeScript Feature

- Dienen nur der Typ-Prüfung
- Erzeugen keinen ECMAScript Code im Output

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

Interfaces

- Interfaces können Variablen und Methoden definieren
- Methoden immer ohne Implementierung
- Member können optional sein

```
interface HasAge {
  birth: Date;
  death?: Date;
  getAge(): number;
}
```

Structural Typing

<> **Typ-System von Java & Co. ist *nominal***
- Kompatibel wenn Erzeuger in gewisser Relation stehen

<> **Typ-System von TypeScript ist *strukturell***
- Typen sind kompatibel, wenn Struktur der erwarteten enspricht
- Erzeuger spielt keine Rolle
- Wird auch als *duck-typing* bezeichnet

Structural Typing


```
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

ECMAScript 6

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

<> Zugriff auf *Member-Variablen* immer über explizites `this`

<> KEIN automatisches `this`-Capturing für Methoden

- Methoden hängen am Prototype
- Ansonsten müssten es Instanz-Variablen sein (schlechtere Performance)

<> Klassen sind ES6-Feature

<> Typisierung über Klassen ist TS-Feature

Hands-on: Klassen

Hands-on

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

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

Konstruktor

ECMAScript 6

<> 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");
```

Hands-on: Konstruktor

Hands-on

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

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

Access Modifier

- Zugriff von "außen" auf Klassen-Eigenschaften kann eingeschränkt werden
- `public`, `private` und `protected`
- Default: `public`
- Reines TypeScript Feature, in JS immer alles `public`

```
class {
  private aMember = 123; // default: public

  private aMethod(): {}
}
```

Konstruktoren & Member-Variablen

<> Gängiger Code

```
class Student {
  name: string;
  matr: number
  constructor(name: string, matr: number) {
    this.name = name;
    this.matr = matr;
  }
}
```

<> Instanzvariablen per Konstruktor-Parameter direkt in Parameterliste anlegen

```
class Student {
  constructor(public name: string, private matr: number) {}
}
```

Readonly Member

<> Instanzvariablen können über `readonly` unveränderbar gemacht werden

<> Müssen direkt oder im Konstruktor gesetzt werden

```
class Student {
  public readonly id: number
  constructor(public readonly name: string, private matr: number) {
    this.id = 42;
  }
}
```

Klassen & Interfaces

<> Werte werden bei Zuweisung per *Structural Typing* verglichen

```
interface Labelled {
    getLabel(): string;
}
let withLabel: Labelled = new Student("Max", 123456);
```

<> Klassen müssen Interfaces nicht explizit implementieren

- Vorteil: Schnittstellen für fremde Klassen definieren

<> Klassen können Interfaces explizit implementieren

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

- Kompabilität schon an Klasse sicherstellen

Vererbung


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

<> **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, ...)`
- Überschreiben von Methoden möglich (hängen am Prototyp)
- Kein Überschreiben von Properties (hängen an Instanz)

<> **Keine Mehrfachvererbung**

Abstract

<> Schlüsselwort `abstract` an Klasse verhindert Instanziierung

<> Methoden können `abstract` sein
- Haben keinen Body
- Erbende Klassen muss implementieren oder `abstract` sein

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

<> Unterschied zu Interfaces: erzeugen ES Code

<> Pro-Tip: Klassen können als Interface verwendet werden

Generics

Generics - Warum

<> **Wiederverwendbarkeit + Typsicherheit**

<> **Datenstrukturen von Typ der Daten unabhängig machen**

- Operationen an Struktur hängen nicht von Typ der Daten ab
  Implementierung von `Array#pop` unabhängig von `string` oder `number`
- Trotzdem wissen was in Datenstruktur steckt
  Was liefert `myArray.pop()`

Generics - Wie

<> Typ-Parameter an Klasse/Interface und Methode

```
interface Array<T> {
  reverse(): T[];

  sort(compareFn?: (a: T, b: T) => number): T[];

  map<U>(
    func: (value: T, index?: number, array?: T[]) => U
  ): U[];
}
```

<> *Type Constraints* werden unterstützt

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

<> Ko- und Kontra-Varianz werden nicht unterstützt

Demo

Generics & Type-Inferencing

Type System Advanced

Object Types

<> Welchen Type hat `obj`?

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

Object Types

<> Object Literale führen zu *object types*

<> Aus Objekt abgeleiteter Typ

```
let obj: { a: number, b: string } = {
  a: 1,
  b: "foo"
};
```

Union Types

<> *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

```
let campusPerson: Student | Professor;

campusPerson = new Student("Max", 123456); // OK
campusPerson = new Professor("Hugo"); // OK
campusPerson = new PostDoc("Seb"); // Fehler
```

Union Types

<> Praktischer Nutzen: Methoden überladen

<> Eine Methode nimmt z.B. `string | string[]` entgegen

<> Innerhalb der Methode Typ-Prüfung

```
1 doSomething(param: string | string[]) {
2   if (Array.isArray(param)) {}
3   else {}
4 }
```

Literal Types

<> Primitive Wert-Literale können als Typen verwendet werden

```
1 let foo: 1; // type: 1, value: undefined
2 foo = 1; // ok
3 foo = 2; // error
```

<> Kann mit Union-Types kombiniert werden

```
1 let unit: "px" | "em" | "%";
2 unit = "px"; // ok
3 unit = "cm"; // error
```

Type Alias mit typeof

<> Typ eines Wertes kann explizit als Typ definiert werden

```
const defaultConfig = {
  foo: false,
  bar: 3
};
type Config = typeof defaultConfig;
```

<> Kann mit Union-Types kombiniert werden

```
const myConfig: Config = {}; // error: foo and bar missing
const myPartialConfig: Partial<Config> = {};

function setConfig(config: Config | Partial<Config>) {}
```

keyof

<> **Szenario**

- Funktion erwartet Name von Property als Parameter
- Wie sicherstellen, dass nur gültige Werte kommen?

<> `keyof` **erzeugt Union-Type aller Property-Namen (String-Literal-Type)**

```
1 function sortBy<T>(t: T[], prop: keyof T): T[] {
2   // sort and return sorted array
3 }
```

Module

Module - Warum

<> Keine Struktur in reinen ES5 Projekten

<> Große Code-Basis == schwieriege Orientierung / Navigation

<> Zu wenig Kapselung / zu viel globales

Module in EcmaScript 5

<> **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

Module - Wie

<> 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

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

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";
```

Hands-on: Module

Hands-on

<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 Statments

<> 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

Top-Level-Module

<> **Module können relativ geladen werden**

- Empfohlen für Code innerhalb eines Projekts

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

<> **Module können auch absolut geladen werden**

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

- Gut für externe Bibliotheken
- Nicht standardisiert von wo geladen wird
- Defacto Standard: `node_modules`

Re-Export

<> 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

Decorators - Wie

<> Ähnlich Annotations in Java und Decorators in Python

<> Transpiler übersetzt Docorator-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 `()`

```
@Freezed()
class Person {

  constructor(@NotUndefined() private name:string) {}

  @LogCall("warn")
  greet() { console.log(`Hello, I'm ${this.name}`); }
}
```

Demo

Decorators

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

```
let x: number;
x;  // Error, reference not preceded by assignment
x = 1;
x; // Ok

let z: number | undefined;
z;  // Ok
```

Null- and undefined-aware types

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

<> Auf `null` prüfen?

<> Auf `undefined` prüfen?

<> Auf beides pürfen?

Null- and undefined-aware types

```
var element = document.querySelector("foo");

element.classList.add("bar"); // error: element can be null

if (element !== null) {
  element.classList.add("bar"); // ok
}
```

async / await

async / await

<> **Asynchrone Programmierung ist schwierig**

- Stark verschachtelter Code
- Oft schlecht lesbar

<> **Async/Await macht Code wieder flach!**

- Teil von ES2017
- Ab TypeScript 2.1 auch für ältere ES Versionen

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

async / await

```
async function askForCleanup() {
  console.log("Please clean up your room");
  while(await !check()) { await doNotGiveIn(); }
  console.log("Thanks!");
}

function check() {
  console.log("Is it cleaned up?");
  return new Promise<boolean>(resolve => resolve(false));
}

function doNotGiveIn() {
  return new Promise<void>(resolve => { console.log("Do it now!"); resolve(); });
}

askForCleanup();
```

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**

<> **Für Angular 2+ quasi gesetzt**
- Module-System von Angular verwendet
- Typ-Informationen gleich dabei
- Gute Unterstützung durch Angular-CLI

<> **Gut in AngulatJS 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