thecodecampus</>

TypeScript Intensive

TypeScript

Inhalt der Schulung

- <> JavaScript Syntax
- <> EcmaScript 6/7 Syntax
- <> TypeScript Basic Syntax
- <> TypeScript OOP
- <> TypeScript Advanced
- <> Modul-Systeme
- <> Tooling
- <> Grundlagen Build-Systeme

Hands-on: Hello World!

<1> Lege die Datei `src/hello.ts` mit folgendem Inhalt an:

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

<2> Compilieren: `./node_modules/.bin/tsc src/hello.ts`

<3> Mit Node ausführen: `node src/hello.js`

this innerhalb Funktionen

<> Gegeben sei folgende Klasse:

```
function Person() {
    this.name = "Max";
    this.sayHello = function () {
        console.log(this.name);
    };
}
```

<> Wie ist die Ausgabe?

```
var p = new Person();
p.sayHello();
```

<> **Hands-on:** Was passiert?

```
var helloFn = p.sayHello;
helloFn();
```

<> **Hands-on**

1. Behebe den Fehler

Arrow Function

<> Kürzere Syntax & this-capturing

<> Syntax normale Funktion: `function (a, b) {...}`

<> Syntax arrow function: `(a, b) => {...}`

<> 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
```

<> Einsatzgebiet: Methoden, Callbacks, Promise-Handler, ...

Structural Typing

<> **Das Type-System von TypeScript ist *structural***
- a.k.a. duck-typing
- Objekte sind kompatibel, wenn deren Struktur gleich ist. Der Name spielt keine Rolle.

<> **Java & Co. sind *nominal***
- Der Name wird zur Prüfung der Kompatibilität verwendet

Beispiel: Structural Typing

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

<> Warum funktioniert das?

Object Types

<> Object **Literale** führen zu *object types*

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

<> Vergleiche die Übung zu Interfaces & Objekte

Union Types

<> *Union Types* sind die Vereinigung von mehreren Typen

<> Syntax

```
let campusPerson: Student | Professor;

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

<> Die Operatoren `||` und `?` erzeugen ebenfalls *union types*

```
let campusPerson = expr ? new Student("Max", 123456) : new Professor("Hugo");
```

Type Guards

<> Warum hat die vorherige Übung funktioniert? Woher wusste der Compiler dass `setSemester(1)` ein gültiger Aufruf ist?

<> Der TypeScript Compiler interpretiert `instanceof` und `typeof a === 'A'` Ausdrücke in Kombination mit `if (...) {...}` (*type guards*)

<> Diese Verhalten kann mit *custom type guards* erweitert werden

<> Syntax:

```
function isBlob(o: any): o is Blob {
  if (...) {
    return true;
  }
  return false;
}
```

Tuple Types

<> Array **Literale** führen zu *tuple types*

```
let tt: [number, string] = [3, "three"];
```

<> Abbildung über *object types*

```
{
  0: number;
  1: string;
}
```

<> Typen sind bei Zugriff bekannt

```
tt[0]; // number
tt[1]; // string
tt[i]; // number | string
```

Specialized Signatures

<> Gängiges Muster in JavaScript: Factory Methoden

```
1 let span = document.createElement("span");
2 let canvas = document.createElement("canvas");
3
4 canvas.height; // Liefert die Höhe des Canvas
5 span.height; // COMPILER FEHLER!
```

<> Woher weiß der **Compiler**, dass `span` kein `height`-Attribut hat?

Specialized Signatures

<> *Specialized Signatures* werden in Interfaces definiert

<> Sind nur für den Compiler relevant

<> Die korrekte Implementierung muss händisch sichergestellt sein

<> Syntax

```
interface Document {
    createElement(tagName: "div"): HTMLDivElement;
    createElement(tagName: "span"): HTMLSpanElement;
    createElement(tagName: "canvas"): HTMLCanvasElement;
    createElement(tagName: string): HTMLElement;
}
```

<> **Wichtig:** Die "normale" (generische) Deklaration muss am Ende stehen

<> Die Rückgabetypen der "spezialisierten" Methoden müssen zuweisungskompatibel zum Rückgabetyp der generischen Methode sein

Generics

<> TypeScript Unterstützt *Generics*

<> *Type constraints* werden unterstützt

<> Co/Kontravarianz werden nicht unterstützt

Generics

<> Syntax Interface/Klasse

```
interface Array<T> {
    reverse(): T[];
    sort(compareFn?: (a: T, b: T) => number): T[];
    // ...
}
```

<> Syntax Methode

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

info@thecodecampus.de
@thecodecampus

www.w11k.de
www.thecodecampus.de

Copyright