

WBE: TYPESCRIPT

JAVASCRIPT MIT TYPENKONZEPT

ÜBERSICHT

- Einführung und Infrastruktur
- Grundlagen von TypeScript
- Funktionen, Klassen, Generics
- Abschliessende Bemerkungen

ÜBERSICHT

- Einführung und Infrastruktur
- Grundlagen von TypeScript
- Funktionen, Klassen, Generics
- Abschliessende Bemerkungen

WAS IST TYPESCRIPT?

- Erweiterung von JavaScript um statisches Typenkonzept
- Dafür: Erweiterung der JavaScript-Syntax
- Type Checking zur Übersetzungszeit
- TypeScript-Compiler übersetzt TypeScript zu JavaScript
- Unterstützung der Entwicklungsumgebung (z.B. VSCode)

<https://www.typescriptlang.org>

VORTEILE VON TYPESCRIPT

- **Typfehler schon beim Kompilieren gemeldet**
(eine Funktion mit String-Parameter kann nicht mit Zahl aufgerufen werden)
- **Bessere Auto-Vervollständigung und Code-Hinweise im Editor**
(für eine Variable vom Typ `string` werden nur String-Methoden angeboten)
- **Eigene Typen können definiert werden**
(flexibles Typenkonzept)

DER TYPESCRIPT COMPILER

```
# Installation im node_modules eines Projekts:  
$ npm install typescript --save-dev  
  
# Starten des Compilers und Hilfe ausgeben  
$ npx tsc  
  
# Anlegen von tsconfig.json mit Default Settings  
$ npx tsc --init
```

tsconfig.json

- Einstellungen für den TypeScript Compiler
- Input- und Output-Verzeichnisse spezifizieren u.a.
- Beispiel unten: ein Aufruf von `tsc` wird dann alle TS-Dateien in `./src` nach `./dist` kompilieren
- Mit `tsc --watch` geschieht dies automatisch beim Speichern

```
{  
  "rootDir": "./src",  
  "outDir":  "./dist"  
}
```

<https://www.typescriptlang.org/tsconfig/>

TYPESCRIPT UND NODE.JS

- In bestimmten Situationen kann TS-Code direkt ausgeführt werden
- Type Stripping (ab Node 23.6.0 automatisch aktiv):
Typ-Annotationen werden entfernt
- Bestimmte TS-Features (z.B. enums) so nicht verarbeitbar, dazu muss Flag `--experimental-transform-types` aktiviert werden ($\geq 22.7.0$)
- Alternativen: Runner wie `ts-node` oder `tsx`,
oder TS Compiler `tsc`

TYPESCRIPT IN WBE

- Praktische Aufgaben können in TypeScript gelöst werden
- Abgabe des durch `tsc` generierten JS-Codes (mit Ausnahme von einem TypeScript-Praktikum)
- In der Prüfung und den Kurztests werden sich die Fragen im Wesentlichen um JavaScript drehen

ÜBERSICHT

- Einführung und Infrastruktur
- Grundlagen von TypeScript
- Funktionen, Klassen, Generics
- Abschliessende Bemerkungen

BASISTYPEN

- Typen: `boolean`, `number`, `string`, `bigint`, `symbol`
- Zuweisung explizit oder implizit (type inference)
- In beiden Fällen ist die Variable vom Typ `string`:

```
let firstName: string = "Dylan"
let lastName = "Miller"

lastName = 33 // Fehler!!
// "Type 'number' is not assignable to type 'string'"
```

TYP `any`

```
let json = JSON.parse("55")

json = true // kein Fehler
json = 33
```

- TypeScript kann den Typ nicht immer herleiten
- In diesem Fall wird der Typ `any` zugewiesen
- Damit ist die Typüberprüfung für diese Variable deaktiviert
- Typ `any` kann auch explizit gesetzt werden

TYP unknown

- Typ `unknown` kann ebenfalls verschiedene Typen annehmen
- Vor bestimmten Operationen aber Type Cast nötig
- Somit sicherer als `any`

```
let w: unknown = {}  
w.ident = () => "hi I'm w" // Fehler!!  
  
let ww = w as { ident: Function }; // Ok  
ww.ident = () => "hi I'm w";
```

ARRAYS

```
const names: string[] = []  
names.push("Dylan")  
names.push(3) // Fehler!!  
  
const numbers = [1, 2, 3] // Typ als number[] erkannt  
numbers.push(4)  
numbers.push("2") // Fehler!!  
  
let head = numbers[0] // head hat Typ number
```

TUPEL (1)

- Arrays fester Länge
- Typ für jede Position festgelegt

```
let ourTuple: [number, boolean, string]  
ourTuple = [5, false, 'Coding God was here']  
  
ourTuple.push('???') // erlaubt, aber sinnvoll?  
  
// oft besser: als readonly Tupel definieren  
const better: readonly [number, number] = [5, 5]
```

TUPEL (2)

- Tupelstellen können benannt werden
(Hinweis zur Bedeutung im Editor / bei Fehlermeldungen)
- Tupel können destrukturiert werden
(wie in JavaScript, sind ja Arrays)

```
const graph: {x: number, y: number} = [55.2, 41.3]  
const [x, y] = graph
```

OBJEKTYPEN

```
const car: { type: string, model: string, year: number } = {  
  type: "Toyota",  
  model: "Corolla",  
  year: 2009  
}
```

- Damit hat `car` genau diese drei Attribute
- Auch die Typen sind für die Attribute festgelegt
- Ohne Typangabe würden die Typen trotzdem so hergeleitet

OPTIONALE ATTRIBUTE

- Optionale Attribute werden mit `?` gekennzeichnet
- Ohne Zuweisung sind sie `undefined`

```
const car: { type: string, mileage?: number } = {  
  type: "Toyota"  
}  
car.mileage = 2000
```

ENUMS (1)

- Ein `enum` repräsentiert eine Gruppe von Konstanten
- Per Default erhält die erste Konstante den Wert 0, dann 1 usw.

```
enum CardinalDirections {  
  North,  
  East,  
  South,  
  West  
}  
let currentDirection = CardinalDirections.North  
console.log(currentDirection) // logs 0
```

ENUMS (2)

- Werte in einem `enum` können auch Strings sein
- In diesem Fall ist die Bedeutung der Werte in der Regel klarer

```
enum CardinalDirections {  
  North = 'North',  
  East = "East",  
  South = "South",  
  West = "West"  
}  
  
console.log(CardinalDirections.North) // logs "North"  
console.log(CardinalDirections.West) // logs "West"
```

<https://www.typescriptlang.org/docs/handbook/enums.html>

TYP ALIAS

- Erlaubt es, Typen einen Namen zu geben
- Möglich für einfache oder komplexe Typen (Objekt, ...)

```
type CarYear = number
type CarType = string
type CarModel = string

type Car = {
  year: CarYear,
  type: CarType,
  model: CarModel
}
```

INTERFACE (1)

- Ein Interface ist ähnlich einem Typ Alias
- Es bietet zusätzliche Möglichkeiten
- Dafür ist es nur für Objekttypen erlaubt

```
interface Rectangle {
  height: number,
  width: number
}

const rectangle: Rectangle = {
  height: 20,
  width: 10
}
```

INTERFACE (2)

```
interface Rectangle {
  height: number
  width: number
}

interface ColoredRectangle extends Rectangle {
  color: string
}
```

- Interfaces können erweitert werden
- `ColoredRectangle` hat alle Attribute von `Rectangle` plus `color`

UNION TYPES

- Eingesetzt, wenn ein Wert mehr als einen Typ haben kann
- Repräsentiert durch den *oder*-Operator `|`

```
let someId: number | string
someId = 1
someId = '2'

let email: string | null = null
email = 'mustepet@zhaw.ch'
email = null

type Id = number | string
let anotherId: Id
anotherId = '1'
anotherId = 2
```

ÜBERSICHT

- Einführung und Infrastruktur
- Grundlagen von TypeScript
- Funktionen, Klassen, Generics
- Abschliessende Bemerkungen

FUNKTIONEN (1)

- Angabe von Parametertypen und Type des Rückgabewerts
- Parameter ohne Typangabe werden als `any` angenommen
- Rückgabebetyp in vielen Fällen per Typinferenz bestimmt
- Ohne Rückgabe wird als Typ `void` angegeben

```
function getTime(): number {
    return new Date().getTime()
}

function printHello(): void {
    console.log('Hello!')
}

function multiply(a: number, b: number) {
    return a * b
}
```

FUNKTIONEN (2)

- Optionale Parameter werden mit `?` markiert
- Parameter können auch Defaults haben

```
function add(a: number, b: number, c?: number) {
    return a + b + (c || 0)
}

function pow(value: number, exponent: number = 10) {
    return value ** exponent
}
```

ÜBERLADEN VON FUNKTIONEN

- Zuerst die Signaturen, dann die Implementierung
- Aufruf mit zwei Argumenten im Beispiel nicht möglich, auch wenn sowohl d als auch y optionale Parameter sind

```
function makeDate(timestamp: number): Date
function makeDate(m: number, d: number, y: number): Date
function makeDate(mOrTimestamp: number, d?: number, y?: number): Date {
    if (d !== undefined && y !== undefined) {
        return new Date(y, mOrTimestamp, d)
    } else {
        return new Date(mOrTimestamp)
    }
}

const d1 = makeDate(12345678)
const d2 = makeDate(5, 5, 5)
```

FUNKTIONSTYPEN

- Auch Funktionstypen können definiert werden
- Diese geben Typen der Parameter und des Rückgabewerts vor

```
type Negate = (value: number) => number
const negateFunction: Negate = (value) => value * -1
```

Mehr zu Funktionen:

<https://www.typescriptlang.org/docs/handbook/2/functions.html>

LITERAL TYPES

```
let changingString = "Hello World" // abgeleiteter Typ: string
const constantString = "Hello World" // abgeleiteter Typ: "Hello World"

function printText(s: string, alignment: "left" | "right" | "center") {
  // ...
}

function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```

- Typ beschränkt auf literale Werte
- Sinnvoll zum Beispiel als Teil einer Union
- Im Unterschied zu Enums sind die Literale hier die Werte

TYPE GUARDS

- Union Types: Operationen müssen alle beteiligten Typen unterstützen
- Einschränkungen auf einzelnen Typ mit Typabfrage in Bedingung

```
type Id = number | string

function swapIdType(id: Id): Id {
  if (typeof id === 'string') {
    // id hat hier sicher den Typ string: String-Operationen möglich
    return parseInt(id)
  } else {
    // id hat hier sicher den Typ number: Zahlen-Operationen möglich
    return id.toString()
  }
}
```

TYPE CASTING

```
let x: unknown = 'hello'
console.log((x as string).length)
console.log((<string>x).length)
```

- Zwei Arten: mit `as` oder mit `<>`
- Variable x wird als String interpretiert
- Achtung: aber keine Umwandlung zu einem String

KLASSEN

```
class Person {  
  private readonly name: string  
  
  public constructor(name: string) {  
    this.name = name  
  }  
  
  public getName(): string {  
    return this.name  
  }  
}
```

- Sichtbarkeit: `public`, `private`, `protected`
- Konstanten: `readonly`

KLASSE IMPLEMENTIERT INTERFACE

```
interface Shape {  
  getArea: () => number  
}  
  
class Rectangle implements Shape {  
  protected readonly width: number  
  protected readonly height: number  
  
  public constructor(width: number, height: number) {  
    this.width = width  
    this.height = height  
  }  
  
  public getArea(): number {  
    return this.width * this.height  
  }  
}
```

KLASSE ERWEITERT KLASSE

```
interface Shape { ... }  
class Rectangle implements Shape { ... }  
  
class Square extends Rectangle {  
  public constructor(width: number) {  
    super(width, width)  
  }  
  
  // getArea gets inherited from Rectangle  
}
```

- Eine Klasse kann mehrere Interfaces implementieren
- Eine Klasse kann eine Klasse erweitern
- Beim Überschreiben von Methoden kann `override` angegeben werden: `public override toString(): string { ... }`

GENERISCHE FUNKTIONEN

- Generics erlauben variable Typen
- Zum Beispiel in Funktionen:

```
function createPair<S, T>(v1: S, v2: T): [S, T] {  
  return [v1, v2]  
}  
  
console.log(createPair<string, number>('hello', 42))  
// ['hello', 42]  
console.log(createPair('hello', 42))  
// same result, infers createPair<string, number>
```

GENERISCHE KLASSEN

```
class NamedValue<T> {  
  private _value: T | undefined  
  constructor(private name: string) {}  
  
  public setValue(value: T) {  
    this._value = value  
  }  
  public getValue(): T | undefined {  
    return this._value  
  }  
  public toString(): string {  
    return `${this.name}: ${this._value}`  
  }  
}  
  
let value = new NamedValue<number>('myNumber')  
value.setValue(10)  
console.log(value.toString()) // myNumber: 10
```

MEHR ZU GENERICS

- Ein Typ Alias kann ebenfalls Typvariablen enthalten
- Das gilt auch für Interfaces
- Typvariablen können auch Defaultwerte haben
- Typvariablen können auch eingeschränkt werden

```
type Wrapped<T> = { value: T }  
const wrappedValue: Wrapped<number> = { value: 10 }  
  
class NamedValue<T = string> { ... }  
let value = new NamedValue('myString')  
let num = new NamedValue<number>('myNumber')  
  
function createValue<S extends string | number>(val: S): Wrapped<S> { ... }
```

UTILITY TYPES

Utility Type	Bedeutung
<code>Partial<OType></code>	alle Attribute des Objekttyps sind optional
<code>Required<OType></code>	alle Attribute des Objekttyps sind notwendig
<code>Record<K, V></code>	Objekttyp mit spezifischem Key und Value Type
<code>Omit<OType, attrs></code>	Objekttyp ohne die Attribute (mit \ getrennt)
<code>Pick<OType, attrs></code>	Objekttyp nur mit den angegebenen Attributen
<code>Exclude<UnionType, types></code>	Union Type ohne die angegebenen Typen
<code>ReturnType<FuncType></code>	Rückgabetyt des Funktionstyps
<code>Parameters<FuncType></code>	Parametertypen des Funktionstyps als Array
<code>Readonly<OType></code>	alle Attribute des Objekttyps sind readonly

KEYOF

- Liefert für einen Objekttyp einen Union Type der verwendetet Keys
- Im Beispiel ist `keyof Person` ein Union Type `"name" | "age"`:

```
interface Person {  
  name: string  
  age: number  
}  
  
function printPersonProperty(person: Person, property: keyof Person) {  
  console.log(`Printing person property ${property}: "${person[property]}"`)  
}
```

NULL

- `null` und `undefined` sind vordefinierte primitive Typen
- Wenn `strictNullChecks` in der Konfiguration gesetzt ist, darf einer Variable nur dann `undefined` oder `null` zugewiesen werden, wenn dies im Typ vorgesehen ist
- Optional Chaining und der Nullish-Coalescence-Operator können wie in JavaScript verwendet werden
- Mit dem `!`-Operator kann angegeben werden, dass ein Wert nicht `null` oder `undefined` ist, obwohl der Typ dies erlauben würde:

```
function getValue(): string | undefined { return 'hello' }  
let value = getValue()  
console.log('value length: ' + value!.length)
```

ÜBERSICHT

- Einführung und Infrastruktur
- Grundlagen von TypeScript
- Funktionen, Klassen, Generics
- Abschliessende Bemerkungen

MERKMALE VON PROGRAMMIERSPRACHEN

- Unterstützte Programmierparadigmen
- Statisches oder dynamisches Typenkonzept
- Werkzeugunterstützung (Editor, IDEs, ...)
- Performanz des erzeugten Codes
- Kompakter Code
- Übersichtlicher Code

PROGRAMMIERPARADIGMEN

Multiparadigmensprachen:

JavaScript, Python, Lisp, ...

Tendenz zu bestimmtem Paradigma:

Java, C++, TypeScript, ...

Ein-Paradigmensprachen:

Haskell, Smalltalk, Prolog, ...

(Abgrenzung nicht immer ganz klar)

STATISCH ODER DYNAMISCH (1)

- **Statisches Typenkonzept (Static Typing):**

Variablen werden mit einem bestimmten Typ deklariert

- explizit
- implizit (Typinferenz)

- **Dynamisches Typenkonzept (Dynamic Typing):**

Der Typ einer Variablen wird zur Laufzeit dynamisch zugewiesen

STATISCH ODER DYNAMISCH (2)

Zur Terminologie: Die Bezeichnungen **statisch/dynamisch** werden teilweise verwechselt oder mindestens nicht klar abgegrenzt von **starker/schwacher** Typisierung

stark / schwach

- Wie leicht lässt sich das Typensystem umgehen?
- **stark:** Java, Smalltalk, Python
- **schwach:** C (u.a. wegen void*)

dynamisch / statisch

- Typ bereits zur Übersetzungszeit festgelegt und unverändert?
- **statisch:** Java, C++
- **dynamisch:** Python, Smalltalk, Lisp

STATISCH ODER DYNAMISCH (3)

Vorteile des statischen Typensystems

- bestimmte Fehler werden bereits beim Übersetzen erkannt
- mehr Unterstützung durch IDEs möglich
- meist performanter

Vorteile des dynamischen Typensystems

- meist schnellere Entwicklung
- meist kompaktere Programme

FEHLER ERKENNEN

- Es ist von Vorteil, Fehler früh zu erkennen
- Typfehler bereits beim Kompilieren erkannt
- Typfehler in Editor/IDE erkannt bzw. verhindert

Das sind grosse Vorteile beim Bau von Software, relativiert allenfalls durch:

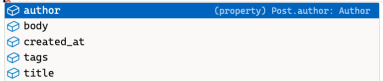
- Es ist anzunehmen, dass Typfehler nicht die problematischsten Fehler in einem Programm sind
- Logikfehler im Wesentlichen durch gründliches Testen gefunden (Test Driven Development?)

WERKZEUGUNTERSTÜTZUNG

Sprachen mit statischem Typenkonzept können mehr Unterstützung bei der Code-Bearbeitung bieten:

- Code-Vervollständigung abhängig von definierten Typen
- Erkennen von Typfehlern

```
12 interface Post {
13   title: string
14   body: string
15   tags: string[]
16   created_at: Date
17   author: Author
18 }
19
20 function createPost(post: Post): void {
21   console.log(`created post ${post} by ${post.author.name}`)
22 }
23
24
25
26
27
```



PERFORMANZ

J. Gosling:

„One of the issues with weak typing systems is they tend to be very hard to get them up to really high performance.”

Bruce Eckel, Why I Love Python (2001):

- Machine performance vs. Programmer Performance
- Most of the time, which is really more important?
- To increase performance, throw hardware at the problem

KOMPAKTER CODE (1)

Beispiel: Conway's Game of Life in APL

```
life ← {>1 ω V.Λ 3 4 = +/ +/ `1 0 1 °.0 `1 0 1 ϕ` Cω}
```

„APL is an array-oriented programming language. Its natural, concise syntax lets you develop shorter programs while thinking more about the problem you're trying to solve than how to express it to a computer.”

https://aplwiki.com/wiki/Main_Page

KOMPAKTER CODE (2)

Kompakter Code führt oft, wenn auch nicht immer (s. APL), zu übersichtlicherem Code:

```
// TypeScript
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
  return arr.map(func)
}

// JavaScript
function map(arr, func) {
  return arr.map(func)
}
```

ÜBERSICHTLICHER CODE

Kopf einer Funktion aus einer Mini-React-Implementierung:

```
const createElement = (  
  type: VirtualElementType,  
  props: Record<string, unknown> = {},  
  ...child: (unknown | VirtualElement)[]  
) : VirtualElement => {
```

TS

```
const createElement = (type, props = {}, ...child) => {
```

JS

FAZIT

- TypeScript vs. JavaScript wird teilweise heftig diskutiert
- Diskussion überlappend mit statischem vs. dynamischem Typensystem
- Beide Ansätze haben je nach Situation mehr Vor- oder mehr Nachteile

Verkürzt und wohl nicht in jeder Situation gültig:

- Weniger Code → weniger Fehler
- Übersichtlicherer Code → weniger Fehler
- Type Check beim Kompilieren → weniger Fehler
- Mehr Unterstützung im Editor → weniger Fehler

Klar ist, dass man mit keinem der Ansätze alle Vorteile bekommt!

VORTEILE VON TYPESCRIPT

- Statisches Typenkonzept: Fehler werden früher und zuverlässiger erkannt
- Zusätzliche Features im Vergleich zu JavaScript
- Open Source, Entwicklung und Support durch Microsoft
- Umfassende IDE-Unterstützung

NACHTEILE VON TYPESCRIPT

- Lernkurve
- Kompilierzeit
- Kleineres Ökosystem (Bibliotheken etc.)
- Mehr Code und in manchen Fällen weniger Übersichtlichkeit

TYPESCRIPT-ALTERNATIVEN

- Im Vergleich zu JS und TS deutlich kleinere Anhängerschaft
- Beide werden wie TypeScript zu JavaScript kompiliert

ReScript

„ReScript is a robustly typed language that compiles to efficient and human-readable JavaScript. It comes with a lightning fast compiler toolchain that scales to any codebase size.”

<https://rescript-lang.org>

ClojureScript

„ClojureScript is a robust, practical, and fast programming language with a set of useful features that together form a simple, coherent, and powerful tool.”

<https://clojurescript.org>

REFERENZEN

- The TypeScript Handbook
<https://www.typescriptlang.org/docs/handbook/intro.html>
- TypeScript Playground
<https://www.typescriptlang.org/play/>
- TypeScript Crash Course (YouTube Tutorial)
<https://www.youtube.com/watch?v=VGu1vDAWNTg&list=PL4cUxeGkcC9gNhFQgS4edYLqP7LkZcFMN>
- TypeScript Tutorial (W3Schools)
<https://www.w3schools.com/typescript/index.php>