

**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>

## Speaker notes

Typen werden zur Übersetzungszeit überprüft.

Im resultierenden JavaScript-Code sind die Type-Annotationen entfernt.

*Type annotations aren't part of JavaScript (or ECMAScript to be pedantic), so there really aren't any browsers or other runtimes that can just run TypeScript unmodified. That's why TypeScript needs a compiler in the first place - it needs some way to strip out or transform any TypeScript-specific code so that you can run it. Most TypeScript-specific code gets erased away, and likewise, here our type annotations were completely erased.*

<https://www.typescriptlang.org/docs/handbook/2/basic-types.html>

# 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
```

## Speaker notes

```
# Globale Installation  
$ npm install -g typescript  
  
# Datei kompilieren  
$ tsc index.ts  
  
# JavaScript ausführen  
$ node index.js
```

# 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/>

## Speaker notes

Es sind zahlreiche Einstellungen möglich, auch mehrere Verzeichnisse mit TS-Dateien:

```
{  
  "include": ["src"],  
  "compilerOptions": {  
    "outDir": "./build"  
  }  
}
```

Auch das Ausführen kann automatisiert werden:

```
node --watch dist/index.js
```

Der TypeScript-Compiler kann Code für verschiedene JavaScript-Versionen erzeugen. Dies ist möglich mit einer target-Angabe im tsconfig oder durch die --target-Option beim Aufruf des Compilers. Ein möglicher Wert ist zum Beispiel es2017.

# 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 (>=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

## Speaker notes

In diesem Fall sind beide Variablen vom Typ any:

```
let age: any  
let title
```

Der Typ any wird manchmal eingesetzt, um Fehlermeldungen zu vermeiden, da es jeden Typ zulässt. Beispielsweise beim Migrieren von JavaScript-Code zu TypeScript kann das manchmal sinnvoll sein, bestimmte Variablen vorübergehend als any zu anzugeben. Typsicherheit ist dann aber nicht mehr gegeben und any sollte schliesslich durch spezifischere Typangaben ersetzt werden.

*Remember, any should be avoided at "any" cost..*

Typinferenz zu any kann mit noImplicitAny als Option im tsconfig.json deaktiviert werden.

Weitere Infos:

- [https://www.w3schools.com/typescript/typescript\\_special\\_types.php](https://www.w3schools.com/typescript/typescript_special_types.php)
- <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#any>

# 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
```

## Speaker notes

Mit der Angabe `readonly` kann ein Array gegen Veränderung geschützt werden:

```
const names: readonly string[] = ["Dylan"]
names.push("Jack") // Fehler!!
```

Ein Array kann auch verschiedene Typen enthalten – in diesem Beispiel ist `things` vom Typ `(number | boolean | string)[]` und `t` vom Typ `(number | boolean | string)` (s. später: Union Types):

```
let things = [1, true, 'hello']
const t = things[0]
```

Ein Array, das beliebige Typen enthalten kann, kann ebenfalls definiert werden:

```
let things: any[] = ['hello', true, 30, null]
things.push({ id: 123 })
```

# 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]
```

## Speaker notes

- Bei der Zuweisung müssen Typen und Länge mit der Tupeldefinition übereinstimmen, ein push auf das Tupel wird aber nicht verhindert
- number [] ist ein beliebig grosses Array von number, [number] dagegen ist ein Tupel bestehend aus einer Zahl

# 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

## Speaker notes

Mit Typinferenz:

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

Nach den Attributangaben in einem Objekttyp können Kommas, Semikolons oder Zeilenwechsel stehen. Das gilt aber nicht für das Objekt selbst. Erlaubt, wenn auch nicht besonders schön:

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

# OPTIONALE ATTRIBUTE

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

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

## Speaker notes

Falls ein Objekt keine feste Liste von Attributen hat, kann auch eine *Indexsignatur* angegeben werden:

```
const nameAgeMap: { [index: string]: number } = {};
nameAgeMap.Jack = 25;           // kein Fehler
nameAgeMap.Mark = "Fifty";     // Fehler!!
```

Indexsignaturen wie diese können auch mit Utility-Typen wie Record<string, number> beschrieben werden.

# 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
```

## Speaker notes

- Werte können auch explizit angegeben werden
- In diesem Fall erhält North den Wert 1, East 2 usw.

```
enum CardinalDirections {  
    North = 1,  
    East,  
    South,  
    West  
}
```

- Alternativ kann auch jeder Konstante ein Wert zugewiesen werden:

```
enum StatusCodes {  
    NotFound = 404,  
    Success = 200,  
    Accepted = 202,  
    BadRequest = 400  
}  
console.log(StatusCodes.NotFound)      // logs 404  
console.log(StatusCodes.Success)       // logs 200
```

# 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>

## Speaker notes

Das Mischen von Zahlen und Strings in einem enum ist nicht empfohlen

# 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
}
```

## Speaker notes

Per Konvention werden Typen mit grossem Anfangsbuchstaben geschrieben.

Hier ist `CarYear` nur ein anderer Name für `number` (eben ein Alias), sie sind dann austauschbar und auch zuweisungskompatibel.

Das Beispiel zeigt, dass man einen Typ Alias nicht nur für Objekte einsetzen kann. Hier ein weiteres Beispiel, wo ein Tupel verwendet wird (Funktionen werden noch behandelt):

```
type Rgb = [number, number, number]

function getRandomColor(): Rgb {
    const r = Math.floor(Math.random() * 255)
    const g = Math.floor(Math.random() * 255)
    const b = Math.floor(Math.random() * 255)

    return [r, g, b]
}

const colorOne = getRandomColor()
const colorTwo = getRandomColor()
console.log(colorOne, colorTwo)
```

# 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  
}
```

## Speaker notes

Wenn im Beispiel bei der Variablen `rectangle` der Typ weggelassen wird, wird nicht automatisch per Inferenz auf den Typ `Rectangle` geschlossen sondern ein Objekttyp mit den beiden Attributen angenommen; trotzdem kann dieser einer `Rectangle`-Variablen zugewiesen werden:

```
let rectangle = {  
    height: 20,  
    width: 10  
}  
  
let r1: Rectangle = rectangle
```

Per Konvention werden Interface-Namen mit grossem Anfangsbuchstaben geschrieben.

Nach den Attributen in einem Objekttyp Alias oder Interface können Kommas oder Semikolons oder keins von beiden stehen. Im letzten Fall ist aber ein Zeilenwechsel nötig.

Hinweis: Bei `type` wird eine Typbeschreibung zugewiesen, es wird der Zuweisungsoperator verwendet. Das ist bei `interface` nicht der Fall.

# 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`

## Speaker notes

Verwendung:

```
const coloredRectangle: ColoredRectangle = {  
    height: 20  
    width: 10  
    color: "red"  
}
```

Neben dem Erweitern von Interfaces mit extends können auch einem bestehenden Interface Attribute hinzugefügt werden:

```
interface Point {  
    x: number  
    y: number  
}
```

```
interface Point {  
    z: number  
}
```

```
let pt: Point = {x: 1, y: 2, z: 3}
```

Dagegen kann ein bestehender Typ nicht geändert werden, nur als Ausgangspunkt für einen weiteren Typ eingesetzt werden:

```
type Point = {  
    x: number  
    y: number  
}  
  
type Point3D = Point & {  
    z: number  
}  
  
let pt2D: Point = {x: 1, y: 2}  
let pt3D: Point3D = {x: 1, y: 2, z:3}
```

# 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
```

## Speaker notes

Bei Union Types ist darauf zu achten, dass mit deren Werten nur Operationen (z.B. Methodenaufrufe) ausgeführt werden, die für alle Typen der Union definiert sind.

# 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
- Rückgabetyp 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
}
```

## Speaker notes

- Funktionsdeklaration mit `function` oder Funktion in Pfeilnotation möglich
- In Pfeilnotation muss auch bei einem Parameter eine Klammer um den Parameter stehen, wenn dieser eine Typangabe hat

```
function addTwoNumbers(a: number, b: number): number {  
    return a + b  
}
```

```
const subtractTwoNumbers = (a: number, b: number): number => {  
    return a - b  
}
```

```
const incr = (n: number) => n+1
```

- Union Type als Parameter:

```
function printStatusCode(code: string | number) {  
    console.log(`My status code is ${code}.`)  
}  
printStatusCode(404)  
printStatusCode('404')
```

- Union Type als Rückgabetyp
- Hier wird automatisch `string | number` als Rückgabetyp abgeleitet:

```
function fun () {
  let x = Math.random()
  if (Math.random() > 0.5) {
    return x
  } else {
    return x.toString()
  }
}
```

- Rückgabe einer Promise ist ebenfalls möglich:

```
async function getCurrentUserId(): Promise<number> {
  // ...
}
```

# 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  
}
```

## Speaker notes

Die Übergabe von Objekten kann ähnlich wie in JavaScript als Ersatz für benannte Parameter verwendet werden:

```
function divide({ dividend, divisor }: { dividend: number, divisor: number }) {
  return dividend / divisor
}

console.log(divide({dividend: 10, divisor: 2}))
```

Auch Rest-Parameter sind – wie in JavaScript – möglich:

```
function add(a: number, b: number, ...rest: number[]) {
  return a + b + rest.reduce((p, c) => p + c, 0)
}
```

# 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

## Speaker notes

Eine Union kann auch andere Typen mit literalen Typen kombinieren:

```
interface Options {
  width: number
}
function configure(x: Options | "auto") {
  // ...
}
configure({ width: 100 })
configure("auto")
```

Hinweis auf ein mögliches Problem: Bei der Typinferenz von Objekten wird von Attributen ausgegangen, welche den Wert ändern können. Im folgenden Beispiel wird für `req.method` der Typ `string` hergeleitet und nicht "GET". Der Aufruf von `handleRequest` ist daher nicht möglich, da "GET" | "POST" verlangt wird:

```
const req = { url: "https://example.com", method: "GET" }

function handleRequest(url: string, method: "GET" | "POST"): void {
  // ...
  return
}

handleRequest(req.url, req.method)
```

Drei Möglichkeiten der Abhilfe:

```
// 1:  
const req = { url: "https://example.com", method: "GET" as "GET" }  
// 2:  
handleRequest(req.url, req.method as "GET")  
// 3:  
const req = { url: "https://example.com", method: "GET" } as const
```

# 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()
  }
}
```

## Speaker notes

- Auf diese Weise die möglichen Typen in einem Programmzweig einzuschränken, nennt man Narrowing
- Ein Union Type kann auch Interfaces enthalten
- In diesem Fall liefert ein `typeof` nicht den gewünschten Typ
- Eine Typangabe als Attribut im Interface kann helfen
- TypeScript kennt eine Reihe weiterer Möglichkeiten

Hier ein paar Beispiele:

```
type Id = number | string

interface User {
  type: 'user'
  username: string
  email: string
  id: Id
}

interface Person {
  type: 'person'
  firstname: string
  age: number
  id: Id
}

function logDetails(value: User | Person): void {
  if (value.type === 'user') {
    console.log(value.email, value.username)
  }
  if (value.type === 'person') {
    console.log(value.firstname, value.age)
  }
}
```

```
function example(x: string | number, y: string | boolean) {  
    if (x === y) {  
        // x und y müssen hier vom Typ string sein  
    } else { ... }  
}  
  
type Fish = { swim: () => void };  
type Bird = { fly: () => void };  
  
function move(animal: Fish | Bird) {  
    if ("swim" in animal) {  
        // animal muss hier vom Typ Fish sein  
    } else { ... }  
}  
  
function logValue(x: Date | string) {  
    if (x instanceof Date) {  
        // x muss hier vom Typ Date sein  
    } else { ... }  
}
```

# 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

## Speaker notes

- Auch Type Assertions genannt
- Sinnvoll, wenn TypeScript den Typ nicht genau erschliessen kann:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement
```

// alternativ:

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_canvas")
```

- Im Beispiel schliesst TypeScript auf HTMLElement, dass es sich um den spezifischeren Typ HTMLCanvasElement handelt, muss man explizit angeben
- Type Casting mit `<>` funktioniert nicht in tsx-Dateien (React)
- Casting Errors können teilweise umgangen werden, indem über unknown gewandelt wird, auch wenn das vermutlich nicht oft sinnvoll ist:

```
(x as unknown) as number
```

- Der folgende Ausdruck sieht mit dem "GET" as "GET" etwas merkwürdig aus
- Der Grund ist, dass method vom Typ "GET" oder "POST" sein soll
- Typinferenz würde den Wert hier aber als string herleiten

```
const req = { url: "https://example.com", method: "GET" as "GET" }
```

# 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`

## Speaker notes

Variablen können auch gleich mit Parametern definiert werden:

```
class Person {  
    // name is a private member variable  
    public constructor(private name: string) {}  
    // ...  
}
```

# 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  
    }  
}
```

## Speaker notes

Eine Klasse kann mehrere Interfaces implementieren, indem sie mit Kommas getrennt nach `implements` angegeben werden.

# 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>
```

## Speaker notes

Weiteres Beispiel:

```
// liefert Typ any
function firstElement(arr: any[]) {
    return arr[0]
}

// für Array von bestimmtem Typ wird dieser Typ geliefert:
function firstElement<Type>(arr: Type[]): Type | undefined {
    return arr[0]
}

// s ist vom Typ string | undefined
const s = firstElement(["a", "b", "c"])
// n ist vom Typ number | undefined
const n = firstElement([1, 2, 3])
// u ist vom Typ undefined
const u = firstElement([])
// v ist vom Typ number | undefined
const v = firstElement([] as 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
```

## Speaker notes

- Instanzen dieser Klasse speichern einen Wert von einem bestimmten Typ
- Wenn der generische Parameter dem Konstruktor übergeben wird (im Beispiel nicht der Fall), kann der Typ auch per Inferenz ermittelt werden
- Ein Array von number kann somit als number [] oder Array<number> geschrieben werden

# 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> { ... }
```

## Speaker notes

Beispiel:

```
function longestTS<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) return a
  else return b
}

const longerArray = longestTS([1, 2], [1, 2, 3])
const longerString = longestTS("alice", "bob")
const notOK = longestTS(1000, 100) // COMPILE ERROR
```

- Hier ist der Typ eingeschränkt: muss ein numerisches length-Attribut haben
- Fehler im Aufruf beim Kompilieren erkannt

Gleiches Beispiel in JavaScript:

```
function longestJS(a, b) {
  if (a.length >= b.length) return a
  else return b
}

const longerArray = longestJS([1, 2], [1, 2, 3])
const longerString = longestJS("alice", "bob")
const notOK = longestJS(1000, 100) // LIEFERT 100 !!!
```

- In JavaScript wird beim letzten Aufruf ein unsinniges Ergebnis zurückgegeben
- Grund: `a.length` ist in diesem Fall `undefined`
- Die Funktionsdefinition mag übersichtlicher sein, aber man muss sicherstellen (oder sicher sein), dass eine Funktion nur mit korrekten Typen aufgerufen wird

# UTILITY TYPES

Utility Typ	Bedeutung
Partial<0Type>	alle Attribute des Objekttyps sind optional
Required<0Type>	alle Attribute des Objekttyps sind notwendig
Record<K, V>	Objekttyp mit spezifischem Key und Value Type
Omit<0Type, attrs>	Objekttyp ohne die Attribute (mit \ \ getrennt)
Pick<0Type, attrs>	Objekttyp nur mit den angegebenen Attributen
Exclude<UnionType, types>	Union Type ohne die angegebenen Typen
ReturnType<FuncType>	Rückgabetyp des Funktionstyps
Parameters<FuncType>	Parametertypen des Funktionstyps als Array
Readonly<0Type>	alle Attribute des Objekttyps sind readonly

## Speaker notes

Beispiel:

```
interface Person {  
    name: string  
    age: number  
    location?: string  
}  
  
const bob: Omit<Person, 'age' | 'location'> = {  
    name: 'Bob'  
    // `Omit` has removed age and location from the type  
}
```

# KEYOF

- Liefert für einen Objekttyp einen Union Type der verwendet 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)
```

## Speaker notes

- Ohne den `!`-Operator im Beispiel würde ein Fehler ausgegeben, dass `value` möglicherweise `undefined` ist
- Wenn `strictNullChecks` gesetzt ist, wird angenommen, dass bei Array-Zugriffen nicht `undefined` zurückgegeben wird (ausser `undefined` ist im Array-Typ explizit enthalten)

# 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**

## Speaker notes

Wo sind TypeScript und JavaScript hier einzuordnen?

# 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

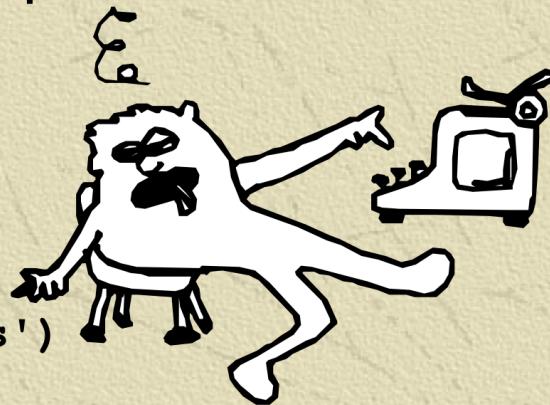
- meist schnellere Entwicklung
- meist kompaktere Programme

Hier noch drei Slides aus der Präsentation *Why I Love Python* von Bruce Eckel (2001). Er hat über 150 Artikel und sechs Bücher verfasst, vor allem zu Programmiersprachen (C++, Java, Python, Scala).

## Weak typing

- Only constraints on an object that is passed into a function are that the function can apply its operations to that object

```
def sum(arg1, arg2):  
    return arg1 + arg2  
  
print sum(42, 47)  
print sum('spam', 'eggs')
```



- “Weak” sounds bad

# Weak Typing in C++: Templates

---

```
#include <string>
#include <iostream>
using namespace std;

template<class A, class B, class R>
R sum(A a, B b) {
    return a + b;
}

int main() {
    string a("one"), b("two")
    cout << sum<string, string, string>(a, b) << endl;
    cout << sum<int, int, int>(1, 2) << endl;
}
```

# Why weak typing isn't weak

---



- ❖ Upcasting becomes meaningless
- ❖ You write *what* you want to do, let Python worry about *how*
- ❖ Argument against weak typing: “errors won’t be found”
  - ◆ Like in pre-ANSI C (had no rules)
- ❖ As long as rules are enforced *sometime*, you’ll find the errors
  - ◆ Heresy: run-time is better than compile time

# 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?)

# WERKZEUGUNTERSTÜZUNG

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
```

A screenshot of a code editor showing code completion for the variable 'post'. The cursor is at the end of 'post.' and a dropdown menu is open, listing the properties of the 'Post' type: 'author', 'body', 'created\_at', 'tags', and 'title'. The 'author' option is highlighted.

# 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 ⍺ v.∧ 3 4 = +/ +/ ¯1 0 1 ∘.⊖ ¯1 0 1 φ.. ⍵}
```

„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](https://aplwiki.com/wiki/Main_Page)

## Speaker notes

Das Gegenteil von kompaktem Code kann man bei Cobol-Programmen sehen. Auch Java wird nachgesagt, viel unnötige Schreibarbeit zu verursachen, wobei sich das im Laufe der Versionen verbessert hat.

# 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!

## Speaker notes

Für grössere Projekte ist die Typsicherheit und Editor-Unterstützung von TypeScript sicher ein deutlicher Vorteil.

In bestimmten Bereichen, zum Beispiel im Bereich Web-Applikationen, haben dynamische Sprachen viele Anhänger. Ein Grund könnte sein, dass wir es etwa im DOM (Document Object Model) oft mit Objekten mit an die 100 Attributen zu tun haben, welche zudem dynamisch ergänzt und bei Bedarf auch wieder entfernt werden.

Neben vielen positiven Stimmen gibt es auch kritische Stimmen zu TypeScript. Zum Beispiel von David Heinemeier Hansson, dem Urheber von Ruby on Rails und bekannt in der Web Community. Er ergründet in einem Blog Post, warum seine Firma von TypeScript zu JavaScript wechselt:

„TypeScript just gets in the way of that for me. Not just because it requires an explicit compile step, but because it pollutes the code with type gymnastics that add ever so little joy to my development experience, and quite frequently considerable grief.“

<https://world.hey.com/dhh/turbo-8-is-dropping-typescript-70165c01>

# 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>

## RESCRIPT

From [rescript-lang.org](https://rescript-lang.org):

*Fast, Simple, Fully Typed JavaScript from the Future*

*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.*

From [ReScript: Rust like features for JavaScript](#):

*If Rust is "C++ in ML clothing" then ReScript is "JavaScript in ML clothing". [...] ReScript has a lightning fast compiler, an easy to learn JS like syntax, strong static types, with amazing features like pattern matching and variant types. Until 2020 it was called "BuckleScript" and is closely related to ReasonML.*

*Rust and ReScript are both strong, statically-typed languages with roots in OCaml. Everything has a type and that type is guaranteed to be correct. Unlike TypeScript you won't find the any type in ReScript.*

From [Difference vs TypeScript](#):

*TypeScript's (admittedly noble) goal is to cover the entire JavaScript feature set and more. ReScript covers only a curated subset of JavaScript. For example, we emphasize plain data + functions over classes, clean pattern matching over fragile ifs and virtual dispatches, proper data modeling over string abuse, etc. JavaScript supersets will only grow larger over time; ReScript doesn't.*

From [rescript-react](#):

*ReScript offers first class bindings for ReactJS and is designed and built by people using ReScript and React in large mission critical React codebases. The bindings are compatible with modern React versions (>= v18.0).*

*No Babel plugins required (JSX is part of the language!)*

From [ReScript React Native](#):

*ReScript React Native is a safe & simple way to build React Native apps, in ReScript, using ReScript React.*

Roots of ReScript:

- **Reason**: Reason lets you write simple, fast and quality type safe code while leveraging both the JavaScript & OCaml ecosystems.  
<https://reasonml.github.io>
- **OCaml**: An industrial-strength functional programming language with an emphasis on expressiveness and safety.  
<https://ocaml.org>
- **Rust**: A language empowering everyone to build reliable and efficient software.  
<https://www.rust-lang.org>

Other links:

- [OCaml Influence Network](#)
- [Rust Influence Network](#)

OCaml was the 2023 winner of the ACM SIGPLAN Programming Languages Software Award:

<https://www.sigplan.org/Awards/Software/>

## CLOJURESCRIPT

ClojureScript is a compiler for Clojure that targets JavaScript. It emits JavaScript code which is compatible with the advanced compilation mode of the Google Closure optimizing compiler.

Clojure is a dynamic, general-purpose programming language supporting interactive development. Clojure is a functional programming language featuring a rich set of immutable, persistent data structures. As a dialect of Lisp, it has a code-as-data philosophy and a powerful macro system.

# 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](https://www.youtube.com/watch?v=VGu1vDAWNTg&list=PL4cUxeGkcC9gNhFQgS4edYLqP7LkZcFMN)
- TypeScript Tutorial (W3Schools)  
<https://www.w3schools.com/typescript/index.php>



