

4 Statisch typisiertes JavaScript mit TypeScript

4.1 Einführung

4.2 Typannotation, -inferenz und grundlegende Typen

4.3 Objekttypen und Interfaces

4.4 Klassen und Namensräume

4.5 Type Assertions und Typkompatibilität

4.6 Generics und fortgeschrittene Typen

4.7 Declaration Merging und weitere Operatoren

TypeScript

Motivation



- Was gibt dieses Programm auf der Konsole aus?

```
const MWST_SATZ = 0.19;

function berechneBruttoBetrag(nettoBetrag) {
  const mwst = nettoBetrag * MWST_SATZ;
  return nettoBetrag + mwst;
}

let nettoBetrag = '100'; // Eingabe via UI
let bruttoBetrag = berechneBruttoBetrag(nettoBetrag);
console.log(bruttoBetrag);
```

TypeScript

Motivation



■ Kleine Ergänzung – große Wirkung

```
const MWST_SATZ = 0.19;

function berechneBruttoBetrag(nettoBetrag: number) {
  const mwst = nettoBetrag * MWST_SATZ;
  return nettoBetrag + mwst;
}

let nettoBetrag = '100'; // Eingabe via UI
let bruttoBetrag = berechneBruttoBetrag(nettoBetrag);
console.log(bruttoBetrag);
```

Fehlermeldung zur Entwicklungszeit

Argument of type 'string' is not assignable to parameter of type 'number'.

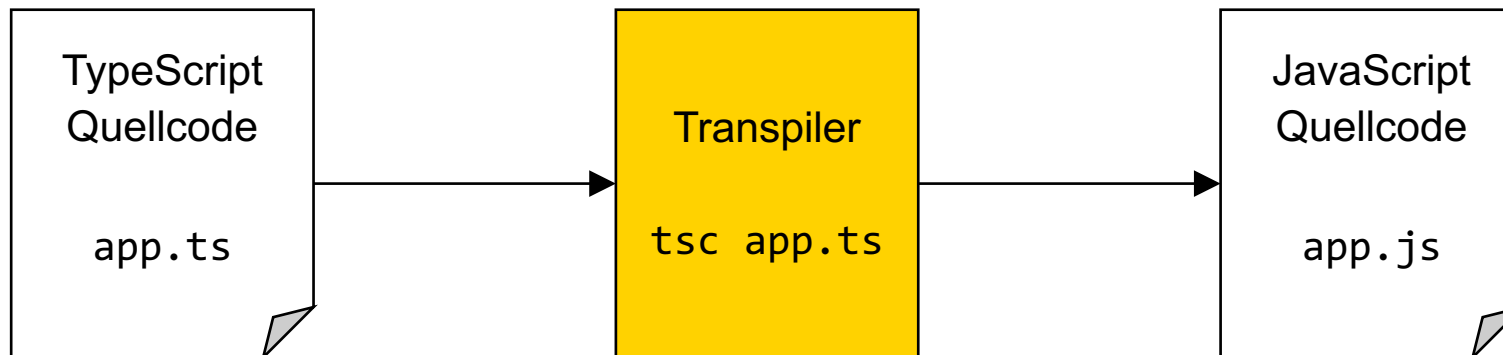
- TypeScript ist eine Obermenge von JavaScript
 - Jedes JavaScript-Programm ist ein gültiges TypeScript-Programm
 - Entwickelt von Microsoft (Open Source)
 - <http://www.typescriptlang.org>
- Hauptmerkmale
 - **Statische Typisierung** mit struktureller Typkonformität („Duck Typing“)
 - Bietet schon jetzt Features aus zukünftigen **ECMAScript-Versionen**
 - Über einen **Transpiler** wird aus einer TypeScript-Datei eine normale JavaScript-Datei erzeugt
- Installation

```
$ npm install -g typescript
```

TypeScript

Transpiler

- Ein Transpiler ist ein spezieller Compiler, der Quellcode einer Programmiersprache in Quellcode einer anderen übersetzt



- tsc ist ein Kommandozeilenprogramm

TypeScript

Beispiel



```
class Person {  
  private displayName: string;  
  constructor(public vorname: string, public nachname: string) {  
    this.displayName = this.vorname + ' ' + this.nachname;  
  }  
  toString() {  
    return this.displayName;  
  }  
}  
let max: Person = new Person('Max', 'Mustermann');  
console.log(max.toString());
```

Datei app.ts

Transpilieren

```
$ tsc app.ts
```

Ausführen (Node.js)

```
$ node app.js
```

TypeScript

Vorteile

- Erhöhung der Code-Qualität und Verständlichkeit durch Einführung statischer Typen
 - Es werden schon zur Entwicklungszeit Fehler durch den Compiler aufgedeckt
 - Typannotationen dokumentieren den Quellcode
- Bietet den Herstellern von Entwicklungsumgebungen die Möglichkeit, Features wie Code-Completion und Refactoring leichter zu entwickeln
 - Kein „JSDoc-Workaround“ mehr nötig
- JavaScript-Features aus zukünftigen ECMAScript-Standards können schon jetzt eingesetzt werden

Konfiguration

Alternativen

- Optionen des tsc Kommandozeilenprogramms

- Beispiel

```
$ tsc --target es6 --module commonjs app
```

- Liste aller Optionen

- <http://www.typescriptlang.org/docs/handbook/compiler-options.html>

- Konfigurationsdatei tsconfig.json

- Eine initiale Konfigurationsdatei kann wie folgt erstellt werden

```
$ tsc --init
```


Konfiguration

Datei tsconfig.json

4 Statisch typisiertes JavaScript mit TypeScript

4.1 Einführung

■ Beispiel

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "strict": true,
    "sourceMap": true
  }
}
```

- Zielversion ist ECMAScript 6
- Modulsyntax ist CommonJS
- Strikte Typüberprüfung wird aktiviert
- Zu jeder .js-Datei wird eine .js.map-Datei erzeugt (für das Debuggen relevant)

- Ein Verzeichnis mit einer `tsconfig.json`-Datei ist das Wurzelverzeichnis eines TypeScript-Projekts
 - Erst wenn diese Datei existiert, werden Abhängigkeiten zwischen den Dateien erkannt
 - Z. B. wird erst dann der Inhalt einer Datei wie `globals.d.ts` berücksichtigt

- Das Projekt wird wie folgt komplett transpiliert

```
$ tsc
```

```
$ tsc -w
```

- Kann im Wurzelverzeichnis des Projekts oder einem beliebigen Verzeichnis unterhalb des Wurzelverzeichnisses (beliebiger Hierarchietiefe) aufgerufen werden
- Mit `-w` wird bei jeder Änderung das Projekt erneut transpiliert (watch-Modus)

Typings

Motivation

- Von JavaScript-Fremdbibliotheken oder von Node.js als Laufzeitumgebung bereitgestellte Variablen, Funktionen etc. sind in TypeScript unbekannt – ihre Nutzung führt zu einem Compile-Error
- Beispiel Node.js

```
const http = require('http');
```

Fehlermeldung zur Entwicklungszeit
Cannot find name 'require'.

Typings

Ambiente Deklarationen

- Das Problem lässt sich mit ambienten Deklarationen lösen
- Eine ambiente Deklaration deklariert eine Variable, Funktion, Klasse, Enum, Namensraum oder Modul, ohne jedoch Auswirkungen auf den generierten JavaScript-Code zu haben
- Beispiel Node.js

```
declare function require(id: string): any;
```

Typings

Definitionsdateien

- Ambiente Deklarationen werden in einer Datei mit der Dateiendung `.d.ts` abgelegt
 - Auch **Definitionsdatei** (*TypeScript Type Definition File*, kurz **Typings**) genannt
- Für eigene Projekte bietet sich folgende Datei an
 - `<path-to-your-project>/globals.d.ts`
 - Wird nur berücksichtigt, wenn die Datei `tsconfig.json` existiert
- Für allgemeine Standardobjekte und Browser-Standardobjekte stehen schon ambiente Deklarationen in der Datei **lib.d.ts** bereit
 - Die Datei wird mit TypeScript installiert und automatisch berücksichtigt
 - Enthält jedoch keine ambienten Deklarationen für Node.js
- Auch Node.js-Module können Definitionsdateien mitliefern

Typings

Installation externer Definitionsdateien

4 Statisch typisiertes JavaScript mit TypeScript

4.1 Einführung

- Nicht jedes Node.js-Paket liefert auch eine Definitionsdatei mit. Auch gibt es zahlreiche Bibliotheken, die nicht als Node.js-Paket vorliegen.
- Es gibt jedoch zentrale Repositories für solche Definitionsdateien
- Die Definitionsdateien können als Node-Pakete installiert werden

```
$ npm i -D @types/node
```

- Suche nach Definitionsdateien: <http://microsoft.github.io/TypeSearch/>

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen**
 - 4.3 Objekttypen und Interfaces
 - 4.4 Klassen und Namensräume
 - 4.5 Type Assertions und Typkompatibilität
 - 4.6 Generics und fortgeschrittene Typen
 - 4.7 Declaration Merging und weitere Operatoren

- Über eine Typannotation wird ein Programmelement explizit (durch den Entwickler) mit Typinformationen versehen

```
function foo(bar: string) { /* ... */ }
```

- In TypeScript ist die Typannotation von Parametern per Default obligatorisch
- Eigenschaften der Konfigurationsdatei hierzu

Eigenschaft	Erlaubte Zuweisungen
strict	Wenn true, dann werden alle Optionen zur strikten Typüberprüfung aktiviert
noImplicitAny	Wenn true, dann wird die Option zur Prüfung impliziter any-Typen aktiviert

Typannotationen

Mögliche Ziele einer Typannotation

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Konstanten und Variablen

```
let anzahl: number;
```

- Funktionsparameter

```
function add(x: number, y: number) { return x + y; }
```

- Rückgabewert einer Funktion

```
function add(x: number, y: number): number { return x + y; }
```

- Der Typ muss nicht immer explizit angegeben werden – vielfach wird er durch **Typinferenz** automatisch ermittelt
- Beispiel: Typ einer Variablen

```
let anzahl = 3;           // Inferierter Typ: number
let farbe;                // Inferierter Typ: any
farbe = 'rot';            // Inferierter Typ nach der Zuweisung: string
```

- Beispiel: Typ des Rückgabewerts einer Funktion (Rückgabetyt)

```
function add(x: number, y: number) { // Inferierter Typ: number
  return x + y;
}
```

Vordefinierte Typen

Typ	Deklaration	Erlaubte Zuweisungen	Fehlerhafte Zuweisungen
boolean	<code>let x: boolean;</code>	<code>x = true;</code> <code>x = false;</code>	<code>x = 42;</code> <code>x = 'foo';</code>
number	<code>let x: number;</code>	<code>x = 42;</code> <code>x = 3.1415;</code> <code>x = 5e2;</code> <code>x = 0xFF;</code> <code>x = Infinity;</code> <code>x = NaN;</code>	<code>x = '42';</code> <code>x = true;</code>
string	<code>let x: string;</code>	<code>x = 'foo';</code> <code>x = "42";</code>	<code>x = 42;</code> <code>x = true;</code>
symbol	<code>let x: symbol;</code>	<code>x = Symbol();</code> <code>x = Symbol('foo');</code>	<code>x = 'foo';</code>
object	<code>let x: object;</code>	<code>x = { foo: 'bar' };</code> <code>x = [1, 2];</code> <code>x = () => 42;</code>	<code>x = 42;</code> <code>x = 'foo';</code>

Vordefinierte Typen

any

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Der Typ `any` definiert keine Einschränkungen – alle Werte sind erlaubt

```
let x: any;  
x = 42;  
x = { foo: 'bar' };
```

- Er ist zu jedem* anderen Typ **zuweisungskompatibel** und umgekehrt

```
let x: any;  
let y: number = 42;  
x = y;      // number ist zuweisungskompatibel zu any  
x = 'foo';  // Typ von x ist weiterhin any  
y = x;      // any ist zuweisungskompatibel zu number
```

*Ausnahme: `any` ist nicht zuweisungskompatibel zu `never`

Vordefinierte Typen

any: strict oder noImplicitAny

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Wird der Parameter **noImplicitAny** oder **strict** auf true gesetzt, dann muss der Typ immer explizit angegeben oder inferierbar sein

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

- Der Rückgabotyp ist inferierbar und muss daher nicht angegeben werden
- Ohne explizite Typangaben sind jedoch die Parametertypen implizit any – dies führt dann zum **Compile-Fehler**

```
function add(a, b) { return a + b; }  
// Fehler: Parameter 'a' implicitly has an 'any' type.  
// Fehler: Parameter 'b' implicitly has an 'any' type
```

```
{  
  "compilerOptions": {  
    /* ... */  
    "strict": true  
  },  
  /* ... */  
}
```

Datei tsconfig.json

Vordefinierte Typen

void

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Der Typ `void` repräsentiert die Abwesenheit eines Wertes und wird als Rückgabebetyp einer Funktion ohne Rückgabewert verwendet

```
function log(msg: string): void {  
    console.log(msg);  
}
```

```
function log(msg: string) {  
    // implizit void (Typinferenz)  
    console.log(msg);  
}
```

- Er ist ein Subtyp von `any` und ist somit zuweisungskompatibel zu `any`

```
let x: any = log('foo');
```

Exkurs

Strikter Null-Checking-Modus

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Dieser Modus ermöglicht es, bestimmte Fehler schon zur Compile-Zeit zu identifizieren
- Beispiel 1

```
let x: number, y: number;  
x = y; // Fehler: Variable 'y' is used before being assigned.
```

- Beispiel 2

```
function div(a: number, b: number) {  
    // inferierter Rückgabotyp: number | undefined  
    if (b !== null) return a / b;  
}  
y = div(3, 4); // Fehler:  
// Type 'number | undefined' is not assignable to type 'number'.
```

Er wird über den Parameter
strictNullChecks oder **strict**
aktiviert



```
{  
  "compilerOptions": {  
    /* ... */  
    "strict": true  
  },  
  /* ... */  
}
```

Datei tsconfig.json

*Wird die Konfigurationsdatei mit `tsc --init` erstellt, dann ist der strikte Null-Checking-Modus aktiviert

Vordefinierte Typen

null und undefined

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Die Typen `null` und `undefined` besitzen jeweils nur einen gültigen Wert: `null` und `undefined`
- Sie sind im **regulären Null-Checking-Modus Subtypen** aller anderen Typen und damit zuweisungskompatibel zu allen anderen Typen

```
let x: number = 42;
let y: object = { };

x = undefined; // Nur im regulären Null-Checking-Modus gültig
y = null;      // s.o.
```

```
{
  "compilerOptions": {
    /* ... */
    "strict": false
  }, /* ... */
}
```

Datei `tsconfig.json`

Vordefinierte Typen

null und undefined

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Im **strikten Null-Checking-Modus** sind sie nur Subtypen von any; zudem ist undefined zuweisungskompatibel zu void

```
let x: any;  
let y: void;  
  
x = null;  
x = undefined;  
y = undefined;  
  
y = null; // Fehler: Type 'null' is not assignable to type 'void'.
```

Vordefinierte Typen

never

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Der Typ **never** repräsentiert den Typ von Werten, die nicht existieren können.

```
function error(message: string): never {  
    throw new Error(message);  
}  
  
function div(x: number, y: number): number {  
    if (y === 0) {  
        return error('division by zero'); // zuweisungskompatibel zu number  
    } else { return x / y; }  
}  
  
let x = div(3, 0);
```

- Er ist Subtyp aller anderen Typen und damit zu diesen zuweisungskompatibel
- Kein Typ ist Subtyp von never; damit ist nur never selbst zu never zuweisungskompatibel (nicht einmal any)

- Ein Enum-Typ definiert eine Menge von numerischen Konstanten oder String-Konstanten

```
enum Farben { Rot, Gruen, Blau }
```

```
enum Farben {  
  Rot = 1, Gruen, Blau  
}
```

```
enum Farben {  
  Rot = 'RED',  
  Gruen = 'GREEN',  
  Blau = 'BLUE'  
}
```

- Wird für eine numerische Konstante kein Wert explizit angegeben, so ist deren Wert der Wert der Vorgängerkonstante + 1 bzw. 0 für die erste Konstante

- Für numerische Konstanten können Ausdrücke angegeben werden

```
enum Farben {  
  Rot = 1,  
  Gruen = Rot << 1, // 2  
  Blau = Gruen << 1 // 4  
}
```

Werte werden beim Transpilieren berechnet

```
enum Farben {  
  Rot = 1 + Math.random() * 10,  
  Gruen = Rot << 1,  
  Blau = Gruen << 1  
}
```

Wert von Rot wird zur Laufzeit ermittelt

- Zusätzlich zum Enum-Typ existiert für jede Konstante ein eigener Typ

```
enum Farben { Rot, Gruen, Blau };  
let farbe: Farben;  
let rot: Farben.Rot = Farben.Rot;  
farbe = rot;  
rot = Farben.Blau; // Fehler
```

- Ein Enum-Typ ist zur Laufzeit eine Variable mit einem Objekt als Wert

```
enum Farben {  
  Rot,  
  Gruen,  
  Blau  
}
```



wird transpiliert zu

```
var Farben;  
(function (Farben) {  
  Farben[Farben["Rot"] = 0] = "Rot";  
  Farben[Farben["Gruen"] = 1] = "Gruen";  
  Farben[Farben["Blau"] = 2] = "Blau";  
})(Farben || (Farben = {}));
```

```
enum Farben {  
  Rot = 'RED',  
  Gruen = 'GREEN',  
  Blau = 'BLUE'  
}
```



wird transpiliert zu

```
var Farben;  
(function (Farben) {  
  Farben["Rot"] = "RED";  
  Farben["Gruen"] = "GREEN";  
  Farben["Blau"] = "BLUE";  
})(Farben || (Farben = {}));
```

- Damit kann anhand eines numerischen Wertes der zugehörige Name der Enum-Konstante ermittelt werden

```
enum Farben { Rot, Gruen, Blau }  
let farbe = Farben.Rot;  
  
console.log(farbe);           // => 0  
console.log(Farben[farbe]);   // => Rot  
console.log(Farben[42]);      // => undefined
```

- Diese Möglichkeit existiert bei Enum-Typen mit String-Konstanten nicht

- Ein mit **const** deklarierter Enum-Typ existiert jedoch nicht zur Laufzeit

```
const enum Farben {  
  Rot, Gruen,  
  Blau }  
let rot = Farben.Rot;
```



wird transpiliert zu

```
let rot = 0 /* Rot */;
```

```
const enum Farben {  
  Rot = 'RED', Gruen = 'GREEN',  
  Blau = 'BLUE' }  
let rot = Farben.Rot;
```



wird transpiliert zu

```
let rot = "RED" /* Rot */;
```

- Somit kann auch nicht anhand eines numerischen Wertes der zugehörige Name der Enum-Konstante ermittelt werden

```
const enum Farben { Rot, Gruen, Blau };  
let farbname = Farben[0]; // Fehler
```

- Numerische Enum-Typen sind zuweisungskompatibel zum Typ `number` und umgekehrt

```
enum Farben { Rot = 1, Gruen, Blau };  
let x: Farben = Farben.Rot; let y: number;  
y = x;  
x = 1; // auch andere Zahlen sind erlaubt, z. B. 42;
```

String-basierte Enum-Typen sind zuweisungskompatibel zum Typ `string` (jedoch nicht umgekehrt)

- Verschiedene Enum-Typen sind untereinander nicht zuweisungskompatibel

```
enum Farben { Rot = 1, Gruen, Blau }  
enum FooBar { Foo = 1, Bar = 2 }  
let x: Farben = Farben.Rot; let y: FooBar = FooBar.Foo;  
y = x; // Fehler
```


- Ein Array-Typ repräsentiert Arrays mit einem gemeinsamen Elementtyp und wird durch ein **Arraytyp-Literal** definiert
- Hierfür gibt es zwei Syntaxalternativen

```
let farben: string[];  
farben = ['rot', 'grün', 'blau'];
```

```
let zahlen: Array<number>;  
zahlen = [3, 4, 5];
```

- Mit Typinferenz

```
let farben = ['rot', 'grün', 'blau']; // inferierter Typ: string[]
```

- Ohne Einschränkung des Typs für die Array-Elemente

```
let werte: any[];  
werte = [42, 'foo', { foo: 'bar' }];
```

Arraytypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Ein Arraytyp S ist zuweisbar zu einem Arraytyp T, wenn der Elementtyp von S zuweisungskompatibel zum Elementtyp von T ist

```
enum Farben { Rot = 1, Gruen, Blau };

let stringArray = ['x', 'y', 'z'];
let numberArray = [1, 2, 3];
let enumArray = [Farben.Rot, Farben.Blau];
let anyArray: any[] = ['x', Farben.Rot, false];

numberArray = enumArray;
enumArray = numberArray;
stringArray = numberArray; // Fehler
anyArray = stringArray;
enumArray = anyArray;
```

- Ein Tupel-Typ repräsentiert Arrays mit individuellen Elementtypen und wird durch ein **Tupeltyp-Literal** definiert

```
let foo: [number, string] = [3, 'drei'];
```

- Ohne Typangabe wird ein Array-Typ inferiert

```
let foo = [3, 'drei']; // inferierter Typ: (string | number)[]
```

- Weniger Elemente sind nicht erlaubt; zusätzlich hingegen schon, müssen jedoch von einem im Tupeltypliteral angegebenen Typ sein

```
let foo: [number, string];  
foo = [3]; // Fehler, da nicht mind. 2 Elemente vorhanden sind  
foo = [3, 'drei', 3]; // OK (zusätzl. Element ist vom Typ number)  
foo = [3, 'drei', true]; // Fehler (true ist weder Zahl noch String)
```

Tupeltypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.2 Typannotation, -inferenz und grundlegende Typen

- Ein Tupeltyp S ist zuweisbar zu einem Tupeltyp T , wenn zu jedem Elementtyp T_j von T ein korrespondierender Elementtyp S_i von S existiert, sodass S_i zuweisungskompatibel zu T_j ist

```
let a: [number] = [41, 42];  
let b: [number, number] = [1, 2];  
let c: [number, string] = [1, 'foo'];  
  
a = b; // OK  
b = a; // Fehler (Typ von a hat zu wenig Komponenten)  
b = c; // Fehler (string ist nicht zuweisungskompatibel zu number)
```

Mikroübung

Richtig oder falsch?



- Der Typ einer Variablen muss immer explizit angegeben werden
- Es gibt keinen Typ, der alle Werte umfasst (Zahlen, Zeichenketten, Objekte etc.)
- Zu bestimmten Enum-Typen existiert zur Laufzeit jeweils ein Objekt

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen
 - 4.3 Objekttypen und Interfaces**
 - 4.4 Klassen und Namensräume
 - 4.5 Type Assertions und Typkompatibilität
 - 4.6 Generics und fortgeschrittene Typen
 - 4.7 Declaration Merging und weitere Operatoren

Funktionstypen

Typdefinition: Funktionstyp-Literal

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein **Funktionstyp-Literal** beschreibt die Parametertypen und den Rückgabetyp der durch den Funktionstyp repräsentierten **Funktionen**

Funktionstyp-Literal

```
let myFunction: (x: number, y: number) => number;
```

- Zuweisung einer Funktion

```
myFunction = (a: number, b: number): number => (a + b) / 2;  
myFunction = (a, b) => (a + b) / 2; // Typinferenz
```

- Parameternamen müssen nicht mit denen des Funktionstyps übereinstimmen
- Parametertypen und Rückgabetyp der zugewiesenen Funktion können inferiert werden

Funktionstypen

Typdefinition: Objekttyp-Literal mit Aufrufsignatur

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Funktionstyp kann alternativ durch ein **Objekttyp-Literal** mit **Aufrufsignatur** definiert werden

Objekttyp-Literal

```
let myFunction: { (x: number, y: number): number };
```

Aufrufsignatur

- Zum Vergleich: Funktionstyp-Literal

```
let myFunction: (x: number, y: number) => number;
```


Funktionstypen

Typdefinition: Interface mit Aufrufsignatur

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Objekttyp-Literal kann durch ein **Interface** benannt werden und ist damit wiederverwendbar

```
interface Operation {  
  (x: number, y: number): number // Aufrufsignatur  
}  
  
let add: Operation;  
let sub: Operation;  
  
add = (x, y) => x + y;  
sub = (x, y) => x - y;  
sub = (x: string, y: number) => x.substr(y); // Fehler
```

Ein Interface ist somit ein benanntes Objekttyp-Literal.

- Im obigen Beispiel wird der **Interface-Typ** (kurz Interface) Operation **deklariert**
- Über den Namen kann der Typ jederzeit referenziert werden (**Typreferenz**)

Funktionstypen

Typinferenz

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Bei einer Variablendeklaration mit direkter Initialisierung muss der Funktionstyp nicht angegeben werden – er kann inferiert werden

```
function avg(a: number, b: number) {  
    return (a + b) / 2;  
}  
let myFunc = avg; // inferierter Typ: (a: number, b: number) => number
```

- Beispiel mit Funktionsausdruck

```
let avg = (a: number, b: number) => (a + b) / 2;  
// inferierter Typ: (a: number, b: number) => number
```

Funktionstypen

Pflichtparameter

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Anders als in JavaScript muss beim Funktionsaufruf für jeden Parameter (per Default) ein Argument übergeben werden; zudem sind zusätzlichen Argumente nicht erlaubt

```
let avg = (a: number, b: number) => (a + b) / 2;  
  
avg(3, 4);    // OK  
avg(3);       // Fehler: 2. Argument fehlt  
avg(3, 4, 5); // Fehler: 3. Argument ist nicht erlaubt
```

- Per Default ist jeder Funktionsparameter ein **Pflichtparameter**

Funktionstypen

Optionale Parameter

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Funktionsparameter können als **optional** deklariert werden

```
let op: (a: number, b?: number) => number; // '?' am Ende des Namens  
// Typ des optionalen Parameters b: number | undefined  
  
op = (a, b?) => b ? a + b : a;  
op = (a, b) => b ? a + b : a; // b ist weiterhin optional
```

- Optionale Parameter müssen nach den Pflichtparametern stehen
- Wird ein Parameter als optional deklariert, so wird der angegebene Typ T erweitert zu T | **undefined**

- Mit Typinferenz

```
let op = (a: number, b?: number) => b ? a + b : a;  
// inferierter Typ: (a: number, b?: number | undefined) => number
```

Funktionstypen

Überladen von Funktionen

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- In JavaScript ist es nicht unüblich, dass der Rückgabebetyp einer Funktion von den Parametertypen abhängt

```
function add(x, y) {  
  if (typeof y === 'number') {  
    return x + y;  
  }  
  else if (Array.isArray(y)) {  
    return y.map(e => x + e);  
  }  
}  
let foo = add(3, 4); // => 7  
let bar = add(3, [4, 5]); // => [7, 8]
```

Funktionstypen

Überladen von Funktionen

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Hierzu können in TypeScript mehrere Funktionstypen zu einer Funktion in Form einer Liste von Überladungen angegeben werden

```
function add(x: number, y: number): number; // Überladung 1
function add(x: number, y: number[]): number[]; // Überladung 2
function add(x: number, y: any): any { // Implementierung
    if (typeof y === 'number') {
        return x + y;
    } else if (Array.isArray(y)) {
        return y.map(e => x + e);
    }
}

let foo = add(3, 4); // Typ von foo: number
let bar = add(3, [4, 5]); // Typ von bar: number[]
let baz = add(3, '4'); // Fehler
```

Zu einem Funktionsaufruf ermittelt der Compiler die **erste passende Überladung** aus der Liste der Überladungen.

Die Überladungen und die Implementierung müssen direkt aufeinander folgen. Nur Leerraum und Kommentar sind dazwischen erlaubt.

Funktionstypen

Überladen von Funktionen

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Die Typangabe für Funktionen mit Überladungen erfolgt über ein Objekttypliteral mit **mehreren Aufrufsignaturen**

```
let op: {  
  (x: number, y: number): number;      // Aufrufsignatur 1  
  (x: number, y: number[]): number[];  // Aufrufsignatur 2  
};  
op = function (x: number, y: any): any {  
  if (typeof y === 'number') {  
    return x + y;  
  } else if (Array.isArray(y)) {  
    return y.map(e => x + e);  
  }  
};  
let foo = op(3, 4);      // Typ von foo: number  
let bar = op(3, [4, 5]); // Typ von bar: number[]  
let baz = op(3, '4');    // Fehler
```

Funktionstypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Funktionstyp S ist zuweisungskompatibel zu einem Funktionstyp T, wenn folgendes gilt
 - 1) Zu jedem **Parametertyp** S_i von S existiert ein korrespondierender Parametertyp T_i von T, sodass T_i zuweisungskompatibel zu S_i ist (**Kontravarianz**)
 - 2) Der **Rückgabetyt** von S ist zuweisungskompatibel zum Rückgabetyt von T (**Kovarianz**)
- Anmerkung zur ersten Bedingung
 - Die Kontravarianz wird nur gefordert, wenn die strikte Funktionstypprüfung aktiviert ist (Parameter strict oder strictFunctionTypes)
 - Andernfalls: **Bivarianz** (T_i ist zuweisungskompatibel zu S_i oder umgekehrt)

```
let foo = (x: number, y: string) => 0;  
let bar = (x: number) => 0;  
let baz = (x: number, y: number) => 0;  
foo = bar; // OK  
bar = foo; // Fehler (1. Bedingung)  
foo = baz; // Fehler (1. Bedingung)
```

```
let foo = (x: [number]) => 0;  
let bar = (x: [number, number]) => 0;  
bar = foo; // OK  
foo = bar; // Fehler (1. Bedingung)
```

```
let foo = (): [number, number] => [1, 2];  
let bar = (): [number] => [42];  
bar = foo; // OK  
foo = bar; // Fehler (2. Bedingung)
```


Konstruktortypen

Typdefinition: Konstruktortyp-Literal

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein **Konstruktortyp-Literal** beschreibt die Parametertypen und den Typ der erzeugten Objekte der durch den Konstruktortyp repräsentierten Konstruktoren

Konstruktortyp-Literal

```
let myConstructor: new (s: string) => Person;  
  
class Person { constructor(public name: string) { } }
```

- Syntax wie beim Funktionstypliteral, jedoch mit Schlüsselwort **new**
- Zuweisung eines Konstruktors

```
myConstructor = Person;  
// Verwendungsbeispiel: let john = new myConstructor('John');
```

Konstruktortypen

Typdefinition: Objekttyp-Literal mit Konstruktsignatur

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Konstruktortyp kann alternativ durch ein **Objekttyp-Literal** mit **Konstruktsignatur** definiert werden

Objekttyp-Literal

```
let myConstructor: { new (s: string): Person };
```

Konstruktsignatur

- Zum Vergleich: Konstruktortyp-Literal

```
let myConstructor: new (s: string) => Person;
```

Konstruktortypen

Typdefinition: Interface mit Konstruktsignatur

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

■ Interface mit Konstruktsignatur

```
class Person { constructor(public name: string) { } }

interface PersonConstructor {
  new (s: string): Person
};

let pc: PersonConstructor;

pc = Person;
pc = class { constructor(public name: string) { } }; // OK
pc = class { constructor(public name: number) { } }; // Fehler
```

Konstruktortypen

Typinferenz

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Bei einer Variablendeklaration mit direkter Initialisierung muss der Konstruktortyp nicht angegeben werden – er kann inferiert werden

```
class Person { constructor(public name: string) { } }  
  
let myConstructor = Person;  
// inferierter Typ: new (s: string) => Person
```

- Beispiel mit Klassenausdruck

```
let myConstructor = class { constructor(public name: string) { } };  
// inferierter Typ: new (s: string) => { n: string }
```

Konstruktortypen

Pflichtparameter und optionale Parameter

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Per Default ist jeder Parameter des Konstruktors ein Pflichtparameter
- Optionale Parameter (wie bei Funktionstypliteralen)

```
class Person {  
    constructor(public name: string, public alter?: number) { }  
}  
  
let myConstructor: new (s: string, n?: number) => Person;  
  
myConstructor = Person;  
// Verwendungsbeispiel: let john = new myConstructor('John');
```

Konstruktortypen

Überladen von Konstruktoren

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Das Überladen von Konstruktoren erfolgt analog zum Überladen von Funktionen

```
class Person {  
  public name: string; public alter: number;  
  
  constructor(name: string, alter: number)           // Überladung 1  
  constructor(params: [string, number])              // Überladung 1  
  constructor(a: any, b?: number) { /* ... */ }      // Implementierung  
}  
  
let myConstructor = Person;  
let john = new myConstructor('John', 42);  
let max = new myConstructor(['Max', 42]);  
let foo = new myConstructor('Max'); // Fehler
```

Konstruktortypen

Überladen von Konstruktoren

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Die Typangabe eines Konstruktors mit Überladungen erfolgt über ein Objekttyp-Literal mit **mehreren Konstruktursignaturen**

```
interface PersonInterface { name: string; alter: number; }
interface PersonConstructor {
  new (s: string, n: number): PersonInterface; // Konstruktursignatur 1
  new (t: [string, number]): PersonInterface; // Konstruktursignatur 2
};

class Person {
  public name: string; public alter: number;
  constructor(a: any, b?: number) { /* ... */ }
}

let myConstructor: PersonConstructor = Person;
let john = new myConstructor('John', 42);
let max = new myConstructor(['Max', 42]);
let foo = new myConstructor('Max'); // Fehler
```

Konstruktortypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Konstruktortyp S ist zuweisungskompatibel zu einem Konstruktortyp T , wenn folgendes gilt
 - 1) Zu jedem **Parametertyp** S_i von S existiert ein korrespondierender Parametertyp T_i von T , sodass T_i zuweisungskompatibel zu S_i ist, oder umgekehrt (**Bivarianz**)
 - 2) Der Typ der Objekte, die von Konstruktoren des Typs S erzeugt werden, ist zuweisungskompatibel zum Typ der Objekte, die von Konstruktoren des Typs T erzeugt werden (**Kovarianz**)

Konstruktortypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

■ Beispiel 1 zur ersten Bedingung

```
let foo = class { constructor(x: number, y: string) { } };  
let bar = class { constructor(x: number) { } };  
let baz = class { constructor(x: number, y: number) { } };  
  
foo = bar; // OK  
bar = foo; // Fehler (1. Bedingung)  
foo = baz; // Fehler (1. Bedingung)
```

■ Beispiel 2 zur ersten Bedingung

```
let foo = class { constructor(x: [number]) { } };  
let bar = class { constructor(x: [number, number]) { } };  
bar = foo; // OK  
foo = bar; // OK (!)
```

Konstruktortypen

Zuweisungskompatibilität

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

■ Beispiel zur zweiten Bedingung

```
let foo = class {  
  x: [number, number];  
  constructor() { }  
};  
  
let bar = class {  
  x: [number];  
  constructor() { }  
};  
  
bar = foo; // OK  
foo = bar; // Fehler (2. Bedingung)  
// [number] ist nicht zuweisungskompatibel zu [number, number]
```

- Ein Objekttyp-Literal/Interface kann allgemein folgende Signaturen enthalten
 - Aufrufsignaturen (*call signatures*)
 - Konstruktsignaturen (*construct signatures*)
 - Eigenschaftssignaturen (*property signatures*)
 - Methodensignaturen (*method signatures*)
 - Indexsignaturen (*index signatures*)

Signaturen

Eigenschaftssignatur

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Objekttyp-Literal/Interface kann über Eigenschaftssignaturen die Eigenschaften von Objekten beschreiben

```
interface Operation {  
  x: number;           // Eigenschaftssignatur  
  y: number;           // s.o. (Eigenschaft y hat den Typ number)  
  eval: () => number;  // s.o. (Eigenschaft eval hat einen Funktionstyp)  
}  
  
let add: Operation;  
  
add = {  
  x: 3,  
  y: 4,  
  eval: function () { return this.x + this.y; }  
}
```

Signaturen

Eigenschaftssignatur



- Eigenschaften können als **optional** und als **readonly** gekennzeichnet werden

```
interface Operation {  
  x: number;  
  y?: number;           // Eigenschaft y ist optional  
  readonly eval: () => number; // Eigenschaft eval ist readonly  
}  
  
let neg: Operation = {  
  x: 10,  
  eval: function () { return -this.x; }  
}  
  
neg.eval = function () { return this.x * this.x; }; // Fehler (readonly)
```

Signaturen

Methodensignaturen

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Eine Methodensignatur ist eine Kurzform für eine Eigenschaftssignatur mit Funktionstyp

```
interface MathFunction {  
    eval(x: number, rep?: number): number;  
    invEval?(x: number): number;  
}  
let sqrt: MathFunction = {  
    eval(a) { return Math.sqrt(a); }  
}
```

- Für die Typüberprüfung sind die Methoden-Parameternamen irrelevant
- Alle Methodenparameter, die nicht explizit als optional gekennzeichnet sind, sind Pflichtparameter
- Eine Kennzeichnung als `readonly` ist nicht möglich

Signaturen

Indexsignaturen

- Ein Objekttyp-Literal mit einer Indexsignatur repräsentiert Array- oder Map-artige Objekte

```
interface StringArray {  
  [index: number]: string;  
}  
  
let x:StringArray = {};  
x[0] = 'Max';  
x[1] = 42; // Fehler!  
x['x'] = 'x'; // Fehler (strict)!  
x.0 = 'foo'; // Fehler!
```

- Alle Eigenschaftswerte müssen vom Typ string sein
- Nur Eigenschaften mit numerischen Namen erlaubt (wenn strict oder noImplicitAny)
- Zugriff nur über Klammer-Notation möglich

```
interface NumberMap {  
  [key: string]: number;  
}  
  
let x: NumberMap = {};  
x['eins'] = 1;  
x.zwei = 2;  
x['3'] = 3;  
x['x'] = 'y'; // Fehler!
```

- Alle Eigenschaftswerte müssen vom Typ number sein
- Zugriff über Klammer- und Punkt-Notation möglich

index und **key** sind nicht vorgegeben – es sind beliebige Bezeichnungen möglich.

Signaturen

Indexsignaturen



- Eine Indexsignatur kann als **readonly** gekennzeichnet werden

```
interface MyReadOnlyMap {  
  readonly [key: string]: any;  
}  
  
let m: MyReadOnlyMap;  
  
m = { a: 1, b: 2 };  
m['a'] = 3; // Fehler, da readonly
```


Signaturen

Trennzeichen

- Mögliche Trennzeichen zwischen Signaturen: Komma und Semikolon
- Es können mehrere Signaturen in einer Zeile deklariert werden
 - Dann müssen diese per Trennzeichen voneinander getrennt werden
- Am Ende einer Zeile kann, muss aber kein Trennzeichen stehen

```
interface Person {  
  vorname: string, nachname: string;  
  geburtsname?: string  
  displayName: () => string  
}
```

Interfaces

Weitere wesentliche Merkmale

4 Statisch typisiertes JavaScript mit TypeScript

4.3 Objekttypen und Interfaces

- Ein Interface existiert nur zur **Entwicklungszeit**
 - Es kann nicht zur Laufzeit geprüft werden, ob ein Objekt ein bestimmtes Interface implementiert (kein `instanceof` möglich)
- Ein Interface kann ein oder mehrere andere Interfaces erweitern

```
interface Student extends Person { matrikelNr: number }
```

- Ein Interface kann von Klassen implementiert werden

```
class StudentImpl implements Student { /* ... */ }
```

Interfaces

Beispiel



```
interface Person {  
  vorname: string;  
  nachname: string;  
}  
function logPerson(person: Person) {  
  console.log(person.vorname + ' ' + person.nachname);  
}  
logPerson({ vorname: 'Max', nachname: 'Mustermann' });
```



wird transpiliert zu

```
function logPerson(person) {  
  console.log(person.vorname + ' ' + person.nachname);  
}  
logPerson({ vorname: 'Max', nachname: 'Mustermann' });
```

- Vergleichen Sie Interfaces in TypeScript mit Interfaces in Java
- Wo sehen Sie Gemeinsamkeiten, wo entdecken Sie Unterschiede?

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen
 - 4.3 Objekttypen und Interfaces
 - 4.4 Klassen und Namensräume**
 - 4.5 Type Assertions und Typkompatibilität
 - 4.6 Generics und fortgeschrittene Typen
 - 4.7 Declaration Merging und weitere Operatoren

Klassen in ECMAScript 6

Definition einer Klasse (Wiederholung)

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Neues Schlüsselwort `class` (ECMAScript 6)
 - Konstruktor hat den Namen `constructor`
 - Statische Methoden werden mit `static` gekennzeichnet
- Syntaktischer Zucker
 - `Person` ist der Konstruktor
 - `toString` ist eine eigene Eigenschaft des Prototyps `Person.prototype`
 - `create` ist eine eigene Eigenschaft der Funktion `Person`

```
class Person {  
  constructor(name) {  
    this.name = name;  
  }  
  toString() {  
    return this.name;  
  }  
  static create(name) {  
    return new Person(name);  
  }  
}  
  
// Erzeugung einer Instanz  
let p1 = new Person('Max');  
console.log(max.toString());  
// Aufruf der statischen Methode  
let p2 = Person.create('Erika');
```

Klassen in ECMAScript 6

Vererbung (Wiederholung)

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Angabe der Oberklasse über das Schlüsselwort `extends`
- Zugriff auf die Oberklasse über das Schlüsselwort `super`
 - Aufruf des Oberklassen-Konstruktors via `super()`
 - Aufruf einer überschriebenen Methode via `super.method()`

```
class Person {  
    constructor(name) {  
        this.name = name;  
    }  
    toString() { return this.name; }  
}  
  
class Student extends Person {  
    constructor(name, matrikelNr) {  
        super(name);  
        this.matrikelNr = matrikelNr;  
    }  
    toString() {  
        return super.toString() +  
            ', ' + this.matrikelNr;  
    }  
}
```

Klassen in TypeScript

Unterschiede zu ECMAScript 6

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Es können auch **Dateneigenschaften** deklariert und initialisiert werden

```
class MyClass { myArray = []; }
```

- Dateneigenschaften können als **readonly** und als **optional** deklariert werden

```
class Octopus {  
  readonly numberOfLegs: number = 8;  
  constructor(readonly name?: string) { }  
}
```

Auch Parameter und Methoden können als optional gekennzeichnet werden (wie bei Objekttypen).

- Solche Dateneigenschaften müssen entweder zusammen mit ihrer Deklaration oder im Konstruktor initialisiert werden
- Bei einem Konstruktorparameter wird damit implizit eine entsprechende Dateneigenschaft deklariert

Klassen in TypeScript

Unterschiede zu ECMAScript 6

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- **Typannotationen** sind möglich (Methodenparameter, Dateneigenschaften, ...)
- **Zugriffsmodifizierer** (`public`, `protected` und `private`)
 - Für Konstruktoren, Methoden, Dateneigenschaften und Parameter der Konstrukturfunktion
 - Default: `public`
 - Ein Konstruktorparameter mit explizitem Zugriffsmodifizierer erzeugt implizit
 - eine Dateneigenschaft mit gleichem Namen und Zugriffsmodifizierer
 - eine Initialisierung der Dateneigenschaft mit dem Parameterwert
- Klassen können ein oder mehrere **Interfaces** implementieren und abstrakt sein

Klassen

Beispiel



```
interface PersonInterface {  
  vorname: string,  
  nachname: string,  
  toString(): string;  
}  
  
class Person implements PersonInterface {  
  private displayName: string;  
  constructor(public vorname: string, public nachname: string) {  
    this.displayName = this.vorname + ' ' + this.nachname;  
  }  
  toString() { return this.displayName; }  
}  
  
let max: PersonInterface = new Person('Max', 'Mustermann');  
console.log(max.toString());
```

Klassen

Abstrakte Klassen

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Klassen können als abstrakt deklariert werden und dann auch abstrakte Methoden und abstrakte Daten-/Zugriffseigenschaften enthalten

```
abstract class Shape {  
  abstract name: string; // alternativ: abstract get name(): string;  
  constructor(public x: number, public y: number) { }  
  abstract area(): number;  
}  
  
class Square extends Shape {  
  constructor(x: number, y: number, public size: number) {  
    super(x, y);  
  }  
  get name() { return 'Square'; }  
  area() { return this.size * this.size; }  
}
```

Klassen

Klassen mit nicht-öffentlichen Konstruktoren

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

■ Beispiel: Singleton

```
class Singleton {  
  private static instance: Singleton;  
  private constructor() { }  
  static getInstance() {  
    if (!Singleton.instance) {  
      Singleton.instance = new Singleton();  
    }  
    return Singleton.instance;  
  }  
}  
  
let obj = Singleton.getInstance();  
let obj2 = new Singleton(); // Error
```

Klassen

Typen



- Eine Klassendeklaration erzeugt neben der Konstruktorfunktion zwei Typen

Klassentyp

- Repräsentiert Instanzen der Klasse
- Nicht-statische Dateneigenschaften und Methoden
- Wenn eine Klasse Interfaces implementiert, dann beziehen sich diese auf den Klassentyp

Konstruktorfunktionstyp

- Repräsentiert die Konstrukturfunktion
- Statische Dateneigenschaften und Methoden

Klassen

Klassentyp



- Eine Klasse kann als Klassentyp wie ein Interface verwendet werden

```
class Person {  
  constructor(public name: string) { }  
  public log() { console.log(this.name); }  
}  
  
let p: Person = { name: 'Max', log: () => { } };  
interface Student extends Person {  
  matrikelNr: number  
}
```

- Ein Objektliteral ist jedoch nur dann typkompatibel zu einer Klasse, wenn deren Dateneigenschaften und Methoden alle public sind

- Zum Konstruktorfunktionstyp kann ein Interface erzeugt werden

```
interface PersonInterface {  
    name: string,  
    toString(): string;  
}  
  
class Person implements PersonInterface {  
    constructor(public name: string) { }  
    toString() { return this.name; }  
    static create(name: string) { return new Person(name); }  
}  
  
interface PersonConstructor {  
    new (name: string): PersonInterface; // Konstruktsignatur  
    create(name: string): PersonInterface;  
}  
  
let pc: PersonConstructor = Person;  
let john = new pc('John Doe');  
let max = pc.create('Max Mustermann');
```

Namensräume

Einleitung

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Namensräume sind eine einfache Alternative zu den ECMAScript 6 Modulen (letztere werden in TypeScript auch unterstützt)
- Beispiel

```
namespace Utility {  
  const ERROR_PREFIX = 'Error: '  
  export function log(msg: string) {  
    console.log(msg);  
  }  
  export function error(msg: string) {  
    console.error(ERROR_PREFIX + msg);  
  }  
}  
Utility.log('...');  
Utility.error('...');  
Utility.ERROR_PREFIX; // Fehler!
```


Namensräume

Merkmale

4 Statisch typisiertes JavaScript mit TypeScript

4.4 Klassen und Namensräume

- Sie ermöglichen eine Strukturierung des Quellcodes und Datenkapselung
 - Strukturierung: Analog zu den Packages in Java
 - Datenkapselung: nur das, was explizit **exportiert** wird, ist außen sichtbar
- Sie erzeugen jeweils eine **globale Variable** (Name = Name des Namensraums)
- Sie können ineinander verschachtelt werden
- Anders als bei den Modulen können Abhängigkeiten zwischen Namensräumen nicht abgebildet werden
- Anders als Module benötigen Namensräume keinen Lademechanismus

Mikroübung

Richtig oder falsch?



- Ein Interface kann eine Klasse erweitern
- Eine Klasse kann eine andere Klasse implementieren
- Namensräume sind wie Interfaces reine Compile-Time-Konstrukte, d. h., sie existieren nicht zur Laufzeit

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen
 - 4.3 Objekttypen und Interfaces
 - 4.4 Klassen und Namensräume
 - 4.5 Type Assertions und Typkompatibilität**
 - 4.6 Generics und fortgeschrittene Typen
 - 4.7 Declaration Merging und weitere Operatoren

Mikroübung

Was ist hier das Problem?

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

```
class Shape { color: string; }
class Circle extends Shape { radius: number; }
class Rectangle extends Shape { width: number; height: number; }

function createShape(kind: string): Shape {
  switch (kind) {
    case 'circle': return new Circle();
    case 'rectable': return new Rectangle();
  }
  throw new Error('Invalid argument: ' + kind);
}

var circle = createShape("circle");
console.log(circle.radius); // Fehler!
```

- Eine Type Assertion ist eine Aussage des Entwicklers über den Typ eines Ausdrucks
 - Sie gilt nur zur Entwicklungszeit und hat keinen Einfluss auf die Laufzeit
 - Wird verwendet, wenn der Entwickler mehr weiß (oder zu wissen glaubt), als der Transpiler mit Sicherheit wissen kann

- Beispiel

```
var circle = createShape("circle") as Circle;
```

- Alternative Syntax

```
var circle = <Circle> createShape("circle");
```

Typkompatibilität

Strukturelle Typkonformität

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

- Typkompatibilität in TypeScript basiert auf der **strukturellen Typkonformität**
- Bei dieser sind die Namen der Typen irrelevant; es kommt nur auf deren Struktur an

```
class Firma { name: string; }  
class Person { name: string; }  
  
let p: Firma;  
p = new Person(); // kein Fehler!
```

- Anmerkung
 - Bei der **nominalen Typkonformität** (z. B. bei Java) sind hingegen auch die Namen der Typen relevant

Typkompatibilität

Strukturelle Typkonformität

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

■ Typisches Beispiel: Aufruf einer Funktion mit einem Objektliteral

```
interface Person { vorname: string, nachname: string };  
function logPerson(person: Person) {  
    console.log(person.vorname + ' ' + person.nachname);  
}  
  
logPerson({ vorname: 'Max', nachname: 'Mustermann' });
```

- Der Typ des Objektliterals (Argument des Funktionsaufrufs) ist strukturell konform zum Typ Person

Typkompatibilität

Strukturelle Typkonformität

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

- Praktischer Nutzen: Eine Klasse muss ein Interface nicht explizit implementieren

```
interface Named {  
    name: string;  
}  
class Mitarbeiter {  
    constructor(public name: string, public abteilung: number) { }  
}  
function logNamed(n: Named) {  
    console.log(n.name);  
}  
let max = new Mitarbeiter('Max Mustermann', 4711);  
logNamed(max);
```

- Die Klasse ist konform zum Interface, wenn Sie zu jedem Pflichtmitglied des Interfaces ein gleichnamiges typkonformes Mitglied definiert

Typkompatibilität

Ausführliche Eigenschaftsüberprüfung

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

- Wird ein **Objektliteral** einer Variablen bzw. einem Funktionsparameter mit Objekttyp zugewiesen, so darf das Objekt nur die im Objekttyp enthaltenen Eigenschaften besitzen

```
interface Person { vorname: string; nachname: string; }  
  
// Fehler, da personalNr unbekannt ist  
let max: Person = { vorname: 'Max', nachname: 'Mustermann',  
                    personalNr: 4711 };
```

- Das gilt auch, wenn der Typ der Variablen über ein Objekttyp-Literal angegeben wird

Typkompatibilität

Ausführliche Eigenschaftsüberprüfung

4 Statisch typisiertes JavaScript mit TypeScript

4.5 Type Assertions und Typkompatibilität

- **Ausweg 1: Zuweisung über eine Zwischenvariable***

```
let p = { vorname: 'Max', nachname: 'Mustermann', personalNr: 4711 };  
let max: Person = p;
```

- **Ausweg 2: Type Assertion***

```
let max: Person = { vorname: 'Max', nachname: 'Mustermann',  
                    personalNr: 4711 } as Person;
```

- **Ausweg 3: Explizites Erlauben anderer Eigenschaften**

```
interface Person {  
  vorname: string, nachname: string, geburtsname?: string  
  [propName: string]: any; // weitere Eigenschaften erlauben  
}
```

*Das Objekt muss weiterhin mindestens die erforderlichen Eigenschaften besitzen

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen
 - 4.3 Objekttypen und Interfaces
 - 4.4 Klassen und Namensräume
 - 4.5 Type Assertions und Typkompatibilität
 - 4.6 Generics und fortgeschrittene Typen**
 - 4.7 Declaration Merging und weitere Operatoren

Generics

Motivation



■ Beispiel: Klasse ValueHolder

```
class ValueHolder {  
  constructor(private value: any) { }  
  getValue() { return this.value; }  
  setValue(value: any) { this.value = value; }  
}  
  
const numberHolder = new ValueHolder(42);  
const value = numberHolder.getValue();
```



Frage: Welchen Typ hat die Variable value und welchen sollte sie haben?

Generics

Motivation



■ Beispiel: Klasse ValueHolder mit Typparameter T

```
class ValueHolder<T> {  
  constructor(private value: T) { }  
  getValue() { return this.value; }  
  setValue(value: T) { this.value = value; }  
}  
  
const numberHolder = new ValueHolder(42);  
const value = numberHolder.getValue();
```



Vorteil: Die Typinformation bleibt erhalten (der Typ der Variablen value ist number)

Generics

Erweiterung des Beispiels

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

- Auch Interfaces und Funktionen können Typparameter deklarieren

```
interface ValueHolder<T> {  
    getValue(): T,  
    setValue(value: T):void;  
}  
  
class SimpleValueHolder<T> implements ValueHolder<T> {  
    constructor(private value: T) { }  
    getValue() { return this.value; }  
    setValue(value: T) { this.value = value; }  
}  
  
function createValueHolder<T>(value: T): ValueHolder<T> {  
    return new SimpleValueHolder(value);  
}  
  
const numberHolder = createValueHolder(42);  
const stringHolder = createValueHolder('Max Mustermann');  
const aString = stringHolder.getValue(); // Typ ist string
```

- Über `extends` kann die Menge der erlaubten Typen für einen Typparameter eingeschränkt werden

```
interface NamedValue { name: string; }  
class Person { constructor(public name: string) {} }  
  
class NamedValueHolder<T extends NamedValue> {  
  constructor(private value: T) { }  
  getValue() { return this.value; }  
  setValue(value: T) { this.value = value; }  
  logValue() { console.log(this.value.name); }  
}  
  
let max = new Person('Max Mustermann');  
let valueHolder = new NamedValueHolder(max);  
valueHolder.logValue();
```

Union-Typen

Motivation

- In zahlreichen JavaScript-Bibliotheken werden Funktionen so implementiert, dass für ihre Parameter alternative Typen möglich sind
- Beispiel Funktion `pick` der `lodash`-Bibliothek

```
var object = { 'a': 1, 'b': '2', 'c': 3 };  
  
_.pick(object, ['a', 'c']); // => { 'a': 1, 'c': 3 }  
_.pick(object, 'a'); // => { 'a': 1 }
```

- Für den zweiten Parameter ist ein Array von Strings oder ein String erlaubt

Union-Typen

Motivation



- Mögliche Implementierung in JavaScript

```
function pick(object, props) {  
  let result = {};  
  if (typeof props === 'string') {  
    result[props] = object[props];  
  } else if (Array.isArray(props)) {  
    props.forEach(prop => result[prop] = object[prop]);  
  }  
  return result;  
}
```

- **Frage:** Wie könnte die TypeScript-Variante hierfür aussehen?

- Ein Union-Typ beschreibt eine Menge von Werten, wobei jeder Wert einen von mehreren alternativen Typen hat
- Beispiel

Union-Type

```
function pick(object: object, props: string | string[]) {  
    // ...  
}
```

- Bei jedem Aufruf von pick ist props **entweder** ein String **oder** ein Array von Strings

Union-Typen

Objektypen

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

- Besteht ein Union-Typ aus Objektypen, so kann direkt (also ohne weitere Maßnahmen) nur auf Eigenschaften zugegriffen werden, die bei allen beteiligten Objektypen existieren

```
interface A { a: string, b: string, c: string };  
interface B { a: string, b: number, d: string };  
  
function foo(x: A | B) {  
  x.a = 'foo';  
  x.b = 'bar'; // Typ von x.b: string/number  
  x.b = 3;      // Erlaubt  
  x.c = 'baz'; // Fehler!  
}
```

- Frage:** Wie kann in dem Beispiel doch auf `x.c` zugegriffen werden?

Union-Typen

Type Guards

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

- Der Union-Typ kann über einen Type Guard eingeengt werden, sodass keine Type Assertions nötig sind
 - Dies ist ein Ausdruck, der zur **Laufzeit** einen bestimmten Typ garantiert
- Beispiel: `typeof` und `instanceof`

```
function pick(object: {[p: string]: any}, props: string | string[]) {  
  let result: {[p: string]: any} = {};  
  if (typeof props === 'string') {  
    // props ist vom Typ string  
    result[props] = object[props];  
  } else {  
    // props ist vom Typ string[]  
    props.forEach(prop => result[prop] = object[prop]);  
  }  
  return result;  
}
```

- Ein Typealias (*type alias*) erzeugt einen neuen **Namen** für einen Typ

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name {
  if (typeof n === "string") {
    return n;
  } else {
    return n();
  }
}
```

- Ein Literal-Typ wird durch ein Literal definiert, welches den einzigen erlaubten Wert dieses Typs darstellt

```
let x: 'foo';  
x = 'foo';  
x = 'bar'; // Fehler
```

Stringliteral-Typ

```
let y: 42;  
y = 42;  
y = 21; // Fehler
```

Numberliteral-Typ

- Literal-Typen werden meist in Verbindung mit Union Types eingesetzt

Literal-Typen

Beispiel

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

```
type Easing = 'ease-in' | 'ease-out' | 'ease-in-out';

function animate(dx: number, dy: number, easing: Easing) {
  switch (easing) {
    case 'ease-in': // ...
    case 'ease-out': // ...
    case 'ease-in-out': // ...
  }
}

animate(10, 10, 'ease-in');
animate(10, 10, 'easein'); // Fehler
```

Discriminated Unions

Motivation: Objektorientierter Ansatz

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

```
abstract class Shape { abstract getArea(): number; }

class Circle extends Shape {
  constructor(public radius: number) { super(); }
  getArea() { return Math.PI * this.radius ** 2; }
}

class Rectangle extends Shape {
  constructor(public width: number, public height: number) { super(); }
  getArea() { return this.width * this.height; }
}

class Square extends Shape {
  constructor(public size: number) { super(); }
  getArea() { return this.size ** 2; }
}

const shape: Shape = new Circle(10);
const area = shape.getArea();
```

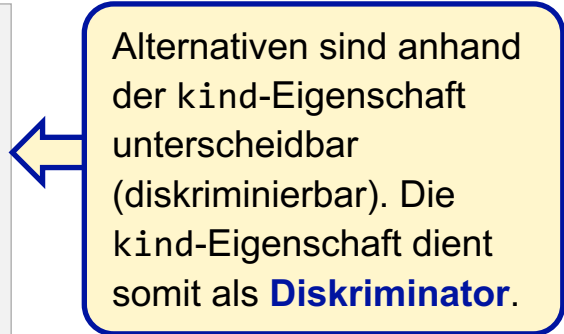

Discriminated Unions

Motivation: Funktionaler Ansatz

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

```
type Shape =  
  { kind: 'circle', radius: number } |  
  { kind: 'rectangle', width: number, height: number } |  
  { kind: 'square', size: number };  
  
function getArea(shape: Shape) {  
  switch (shape.kind) {  
    case 'circle':    return Math.PI * shape.radius ** 2;  
    case 'rectangle': return shape.width * shape.height;  
    case 'square':    return shape.size ** 2;  
  }  
  throw new Error('Invalid shape');  
}  
  
const shape: Shape = { kind: 'circle', radius: 10 };  
const area = getArea(shape);
```



Alternativen sind anhand der kind-Eigenschaft unterscheidbar (diskriminierbar). Die kind-Eigenschaft dient somit als **Diskriminator**.

Discriminated Unions

■ Bestandteile eines Discriminated Union-Typs

```
type Shape =  
  { kind: 'circle', radius: number } |  
  { kind: 'rectangle', width: number, height: number } |  
  { kind: 'square', size: number };
```

- Mehrere Objekttypen mit einer **gemeinsamen Eigenschaft**, deren Typ ein Stringliteral-Typ ist (**Diskriminator-Eigenschaft**)
 - Vereinigung dieser Objekttypen durch einen Union-Typ
 - Typalias, um den Union-Typ einen Namen zu geben
- ## ■ Type Guard ermöglicht Typeinengung anhand der Diskriminator-Eigenschaft

```
switch (shape.kind) { case 'circle': // Typ ist eingeengt /*... */ }
```

Diskutieren Sie zu zweit folgende Fragen:

- Welche Vorteile ergeben sich aus der Verwendung von Discriminated Unions (funktionaler Ansatz) im Vergleich zum objektorientierten Ansatz mit dedizierten Klassen?

Intersection-Typen

- Ein Intersection-Typ beschreibt eine Menge von Werten, wobei jeder Wert alle angegebenen Typen hat
- Beispiel

Intersection-Type

```
type Enjoyable = Drinkable & Eatable ;

interface Drinkable { drink: () => void; }
interface Eatable { eat: () => void; }

var x = {
  drink: () => { console.log('drink'); },
  eat: () => { console.log('eat'); }
}
var y = { drink: () => { console.log('drink'); } }
var e: Enjoyable = x;
e = y; // Fehler
```

Type Queries

typeof

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

- Über eine Type Query kann der Typ bestimmter Ausdrücke ermittelt werden, um z. B. eine Variable gleichen Typs zu deklarieren

```
let foo = { x: 10, y: 'bar' };  
let baz: typeof foo; // Typ: { x: number, y: string }
```

- Unterstützte Ausdrücke
 - Namensreferenzen (*identifier reference*)
 - Namen von Namensräumen, Klassen, Variablen etc.
 - Eigenschafts-Zugriffs-Ausdruck (*property access expression*), z. B.

```
let foo = { x: 10, y: { a: 42, b: ['bar'] } };  
let baz: typeof foo.y.b; // Typ: string[]
```

Type Queries

typeof



■ Resultierender Typ bei Namensreferenzen

Ausdruck	Typ
Namensraum	Objekttyp der Namensrauminstanz
Klasse	Konstruktortyp der Konstruktorfunktion
Enum	Objekttyp des Enum-Objekts
Funktion	Funktionstyp der Funktion
Variable	Typ der Variablen
Parameter	Typ des Parameters

Type Queries

keyof



- Eine Index Type Query **keyof** **T** liefert den Typ der erlaubten Eigenschaftsnamen für T (keyof T ist somit ein Subtyp von string)

```
interface Person {  
  vorname: string,  
  nachname: string,  
  alter?: number  
}  
  
let foo: keyof Person; // Typ: "vorname" | "nachname" | "alter"  
  
foo = 'vorname';  
foo = 'alter';  
foo = 'geburtsname' // Fehler;
```

Type Queries

T[K]



- Über den Indexed-Access-Operator T[K] kann der Typ einer Objekteigenschaft anhand eines Stringliteral-Typs ermittelt werden

```
interface Person {  
  vorname: string,  
  nachname: string,  
  alter?: number  
}  
  
let foo: Person['vorname'];           // Typ: string  
let bar: Person['alter'];             // Typ: number | undefined  
let baz: Person['vorname' | 'alter']; // Typ: string | number | undefined
```

```
type Foo = String['toLowerCase']; // Typ: () => string
```


Type Queries

Beispiel



```
interface Person {  
  vorname: string,  
  nachname: string,  
  alter?: number  
}  
  
function pluck<T, K extends keyof T>(o: T, names: K[]): T[K][] {  
  return names.map(n => o[n]);  
}  
  
let person: Person = {  
  vorname: 'John',  
  nachname: 'Doe',  
  alter: 42  
};  
  
let x: string[] = pluck(person, ['vorname', 'nachname']); // ['John', 'Doe']  
let y: string[] = pluck(person, ['vorname', 'alter']);    // Fehler
```

Mapped Types

Motivation

- Häufig wird zu einem Objekttyp eine Variante benötigt, bei der alle Eigenschaften optional oder readonly sind

```
interface Person {  
  name: string  
  alter: number  
}
```

```
interface PersonPartial {  
  name?: string  
  alter?: number  
}
```

```
interface PersonReadonly {  
  readonly name: string  
  readonly alter: number  
}
```

- Ein Mapped Type wird auf der Basis eines anderen Typs gebildet, indem die Properties transformiert werden

```
interface Person { name: string, alter: number }

type PersonReadOnly = {
  readonly [P in keyof Person]: Person[P];
}

type PersonPartial = {
  [P in keyof Person]?: Person[P];
}

let john: PersonPartial = { name: 'John' };
let max: PersonReadOnly = { name: 'Max', alter: 42 };
max.alter = 43; // Fehler
```

- Durch Einsatz von Typparametern lassen sich allgemeine Typabbildungen definieren

```
type Readonly<T> = { readonly [P in keyof T]: T[P]; }  
type Partial<T> = { [P in keyof T]?: T[P]; }  
  
interface Person { name: string, alter: number }  
  
type PersonPartial = Partial<Person>;  
type ReadonlyPerson = Readonly<Person>;
```

Die Typen `Readonly<T>` und `Partial<T>` gehören zur **Standardbibliothek** von TypeScript. Sie dürfen nicht erneut definiert werden.

Mapped Types

Typen der Standardbibliothek von TypeScript

4 Statisch typisiertes JavaScript mit TypeScript

4.6 Generics und fortgeschrittene Typen

- Readonly, Partial (siehe vorherige Folie)
- Pick

```
type Pick<T, K extends keyof T> = {[P in K]: T[P]; }  
  
interface Person { vorname: string, nachname: string, alter: number }  
type PersonShort = Pick<Person, 'vorname' | 'nachname'>;  
let john: PersonShort = { vorname: 'John', nachname: 'Doe' };
```

- Record

```
type Record<K extends string, T> = { [P in K]: T; }  
  
type Person = Record<'vorname' | 'nachname', string>;  
let john: Person = { vorname: 'John', nachname: 'Doe' };
```

- 4 Statisch typisiertes JavaScript mit TypeScript
 - 4.1 Einführung
 - 4.2 Typannotation, -inferenz und grundlegende Typen
 - 4.3 Objekttypen und Interfaces
 - 4.4 Klassen und Namensräume
 - 4.5 Type Assertions und Typkompatibilität
 - 4.6 Generics und fortgeschrittene Typen
 - 4.7 Declaration Merging und weitere Operatoren**

Declaration Merging

Einleitung

4 Statisch typisiertes JavaScript mit TypeScript

4.7 Declaration Merging und weitere Operatoren

- TypeScript ermöglicht es über das *Declaration Merging*, bestehende Interfaces, Klassen, Namensräume etc. zu ergänzen
 - Allgemein werden beim Declaration Merging zwei oder mehrere Deklarationen zu einer kombiniert, die die Merkmale aller Deklarationen besitzt
- Nutzungsmöglichkeiten (Auswahl)
 - Ein bestehendes Interface um neue Eigenschaften erweitern
 - Einen bestehenden Namensraum um neue Deklarationen erweitern
 - Deklaration innerer Klassen
 - Erweiterung einer bestehenden Funktion um Dateneigenschaften

Declaration Merging

Beispiel: Interfaces

4 Statisch typisiertes JavaScript mit TypeScript

4.7 Declaration Merging und weitere Operatoren

```
interface Box {  
  height: number;  
  width: number;  
}  
  
interface Box {  
  scale: number;  
}  
  
let box: Box = { height: 5, width: 6, scale: 10 };
```


Declaration Merging

Beispiel: Namensräume

4 Statisch typisiertes JavaScript mit TypeScript

4.7 Declaration Merging und weitere Operatoren

```
namespace Utility {  
  const ERROR_PREFIX = 'Error: ';  
  export function error(msg: string) {  
    console.error(ERROR_PREFIX + msg);  
  }  
}  
  
namespace Utility {  
  // hier ist ERROR_PREFIX nicht sichtbar  
  export function log(msg: string) {  
    console.log(msg);  
  }  
}  
Utility.log('...');  
Utility.error('...');
```

Declaration Merging

Beispiel: Innere Klassen

4 Statisch typisiertes JavaScript mit TypeScript

4.7 Declaration Merging und weitere Operatoren

```
class Album {  
  constructor(label: Album.AlbumLabel) { };  
}  
  
namespace Album {  
  export class AlbumLabel {  
    constructor(public label: string) { }  
  }  
}  
  
var a = new Album(new Album.AlbumLabel('...'));
```

Declaration Merging

Beispiel: Funktionen ergänzen

4 Statisch typisiertes JavaScript mit TypeScript

4.7 Declaration Merging und weitere Operatoren


```
function buildLabel(name: string): string {  
    return buildLabel.prefix + name + buildLabel.suffix;  
}  
  
namespace buildLabel {  
    export let suffix = '';  
    export let prefix = 'Hallo ';  
}  
  
console.log(buildLabel('Max Mustermann'));
```

■ Non-null Assertion Operator

```
interface Entity { name: string; }

function validateEntity(e?: Entity) {
  if (!e) { throw new Error(); }
  else    { return !!e.name; }
}

function processEntity(e?: Entity) {
  validateEntity(e);
  let s = e!.name; // Non-null-Assertion-Operator: !
}
```



Dieser Operator ist nur relevant, wenn der strikte Null-Checking-Modus aktiviert ist.

- Informiert TypeScript darüber, dass der Operand (hier: e) weder null noch undefined ist
- Hat keine Auswirkung auf den generierten JavaScript-Code

■ Spread-Operator für Objekte

```
let foo = { x: 1, y: 2 };  
let bar = { x: 0, ...foo, y: 3, z: 4 }; // Spread-Operator: ...  
console.log(bar); // { x: 1, y: 3, z: 4 }
```

■ Rest-Operator für Objekte

```
let foo = { x: 1, y: 2, z: 3 };  
let { z, ...bar } = foo; // Rest-Operator: ...  
console.log(z); // 3  
console.log(bar); // { x: 1, y: 2 }
```