



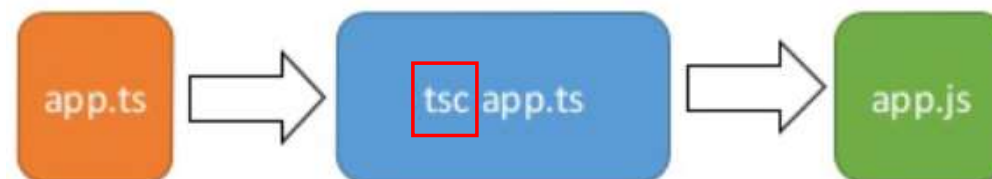
Web Frameworks

TypeScript

Bernhard Löwenstein

TypeScript vs. JavaScript

- TypeScript bildet eine Obermenge von JavaScript und erweitert JavaScript um bestimmte Features.
- Der TypeScript-Compiler (tsc) transpilet den Code in JavaScript-Code.



Features von TypeScript

- TypeScript bietet folgende Features:
 - strenge Typisierung
 - Typinferenz
 - Klassen, klassenbasierte Vererbung und Dekoratoren
 - Typparameter (Generics)
 - abstrakte Schnittstellentypen (Interfaces)
 - zusätzliche Typen: *any*, *void*, Tupel, Funktionstypen, Vereinigungstypen (Union), Kreuzungstypen (Intersection) und Aufzählungstypen (*enum*)
 - Mehrfachvererbung mit Mixins
 - Typalias
 - Namensräume

Node.js

- Node.js ist eine plattformübergreifende Open Source-Laufzeitumgebung für serverseitiges JavaScript.
- Node.js ist erforderlich, um JavaScript (ohne Browserunterstützung) auszuführen.

<https://nodejs.org>

IDE-Unterstützung

- TypeScript wird von einer Vielzahl von Entwicklungsumgebungen unterstützt, z.B. Brackets, Eclipse, IntelliJ IDEA, Visual Studio Code oder WebStorm.

Kernfeatures

Strenge Typisierung

- Eines der zentralen Features von TypeScript ist die strenge Typisierung von Variablen und Parametern.

Vorteil der strengen Typisierung

- Die IDE bzw. der Compiler kann Fehler erkennen.

```
let isVisible: boolean = true;  
isVisible = "hidden"; // Error
```

TS2322: Type '"hidden"' is not assignable to type 'boolean'.

[Remove unused assignment](#)

Alt+Umschalt+Eingabe

[More actions...](#)

Alt+Eingabe

Datentyp *boolean*

- Der Typ wird nachgestellt, also nach dem Variablennamen angegeben.

```
let isVisible: boolean = true;  
const hide: boolean = false;
```

- Mit *let* und *const* deklarierte Variablen sind nur in dem Block verfügbar, in dem sie deklariert wurden.

Datentyp *number*

- Da TypeScript eine Obermenge zu JavaScript darstellt, gibt es keine unterschiedlichen Zahlentypen (wie z.B. *int* oder *double* in Java).

```
let width: number = 1234;
```

Datentyp *string*

- Wie in JavaScript lassen sich Strings mittels einfacher bzw. doppelter Anführungszeichen sowie mittels Backticks erstellen.

```
let firstName: string = "Hans";
```

```
let lastName: string = 'Peters';
```

```
let fullName = firstName + ' ' + lastName;
```

Arrays

- Weiterhin kann man Arrays definieren – entweder in dieser Form

```
let list: number[] = [1, 2, 3];
```

oder in der generischen Variante:

```
let list: Array<number> = [1, 2, 3];
```

Arrays

- Der Zugriff auf die Elemente eines Arrays erfolgt wie folgt:

```
let firstnames: string[] = ["Julia", "Anna", "Thomas"];

if (firstnames.length > 0) {
    console.log("First Entry: " + firstnames[0]);
}
```

Arrays

- Das Durchlaufen eines Arrays erfolgt wie folgt:

```
let firstnames: string[] = ["Julia", "Anna", "Thomas"];  
for (let firstname of firstnames) {  
    console.log(firstname);  
}
```

Tupel

- Ein Tupel ist ein Array-ähnlicher TypeScript-Typ, allerdings mit ein paar Besonderheiten:
 - Die Anzahl der Elemente ist festgelegt.
 - Die Art der Elemente ist bekannt.
 - Die Typen der Elemente müssen nicht identisch sein.

Tupel

- Tupel deklarieren:

```
let contact: [string, string, number];
```

- Tupel initialisieren:

```
contact = ["Herbert", "Peters", 27];
```

- Tupel auslesen und setzen:

```
let firstName: string = contact[0];  
firstName = "Hubert";  
contact[0] = firstName;
```

Datentyp *enum*

- Aufzählungstypen werden mittels *enum* wie folgt definiert und verwendet:

```
enum Color {Red, Green, Blue}  
let c: Color = Color.Green;
```

Datentyp *any*

- Soll der Datentyp einer Variable nicht fix festgelegt werden, kann das mittels *any* ausgedrückt werden.

```
let isVisible: any = true; // OK  
isVisible = "hidden"; // OK  
isVisible.IrgendeineMethode(); // OK
```

Datentyp *object*

- Die Basis für alle nicht-primitiven Datentypen ist *object* (vgl. Klasse *Object* in Java).
- Ein beliebiges Objekt kann mithilfe eines Objekt-Literals definiert werden.

```
let myTyp = {  
  myStatus: true,  
  lastName: "Müller"  
}
```

Datenkapselung

- Mit den Zugriffsmodifiern (access modifier) wird festgelegt, auf welche Eigenschaften, Methoden und Konstruktoren andere Klassen zugreifen können.
- Es gibt in TypeScript folgende Modifier:
 - *public*
 - *private*
 - *protected*

Zugriffsmodifizier *public*

- Standardmäßig sind alle Mitglieder einer Klasse *public*.
- Auf solche Mitglieder kann von überall ohne Einschränkungen zugegriffen werden.

```
class Employee {  
    public empCode: string;  
    empName: string;  
}
```

```
let emp = new Employee();  
emp.empCode = "123";  
emp.empName = "Franz";
```

Zugriffsmodifizier *private*

- Beim Modifizier *private* sind Zugriffe nur innerhalb der Klasse möglich.

```
class Employee {  
    private empCode: number;  
    empName: string;  
  
    constructor() {  
        this.empCode = 456;  
    }  
}
```

```
let emp = new Employee();  
emp.empName = "Franz";  
emp.empCode = 123;
```

TS2341: Property 'empCode' is private and only accessible within class 'Employee'.

Make field 'empCode' public Alt+Umschalt+Eingabe

More actions... Alt+Eingabe

Zugriffsmodifizier *protected*

- Der Modifizier *protected* ähnelt *private*, allerdings können abgeleitete Unterklassen auf solche Mitglieder zugreifen.

```
class Employee {  
    public empName: string;  
    protected empCode: number;  
  
    constructor(name: string, code: number){  
        this.empName = name;  
        this.empCode = code;  
    }  
}  
  
class SalesEmployee extends Employee{  
    private department: string;  
  
    constructor(name: string, code: number, department: string) {  
        super(name, code);  
        this.department = department;  
  
        this.empCode = 456;  
    }  
}  
  
let emp = new SalesEmployee( name: "John Smith", code: 123, department: "Sales");  
emp.empCode;
```

TS2445: Property 'empCode' is protected and only accessible within class 'Employee' and its subclasses.

[Make field 'empCode' public](#) Alt+Umschalt+Eingabe [More actions...](#) Alt+Eingabe

Getter und Setter

```
class Employee {  
    private _empCode: string;  
  
    get empCode(): string {  
        return this._empCode;  
    }  
  
    set empCode(empCode: string) {  
        this._empCode = empCode;  
    }  
}
```

Getter und Setter

- Aufgrund der Syntax ist beim Zugriff von außen nicht erkennbar, ob intern Getter bzw. Setter verwendet werden.

```
const employee: Employee = new Employee();
```

```
employee.empCode = "new code";
```

```
console.log(employee.empCode);
```

Datentyp *void*

- Der Typ *void* wird als Rückgabebetyp für Methoden genutzt, die nichts zurückliefern.

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

Datentyp *undefined* und *null*

- In TypeScript haben diese beiden Werte eigene Typen und sind standardmäßig Untertypen von allen anderen Typen.

```
let firstName: string = "Thomas";  
firstName = null; // OK  
firstName = undefined; // OK
```

Datentyp *undefined* und *null*

- Wenn eine Variable *undefined* oder *null* ist, gibt es einen Laufzeitfehler, wenn versucht wird auf ein Mitglieder der Variable zuzugreifen.

```
let firstName: string = null;  
let firstNameLength: number = firstName.length;
```

firstName is null



Interfaces

- Ein Interface spezifiziert die Eigenschaften und Methoden eines Objekts, implementiert die Methoden aber nicht.

Klassen

- Eine Klasse beschreibt, wie ein Objekt aussehen und sich verhalten soll.
- Von einer Klasse können Instanzen erzeugt werden. Diese Objekte besitzen Eigenschaften, Methoden und maximal einen Konstruktor.

Interfaces vs. Klassen

- Im Gegensatz zur Klasse ist ein Interface eine virtuelle Struktur, die nur im Kontext von TypeScript vorhanden ist.
- Von Interfaces können keine Instanzen erzeugt werden.
- Der TypeScript-Compiler verwendet Interfaces ausschließlich zur Typüberprüfung. Sobald der Code in JavaScript kompiliert wird, werden Interfaces ausgelöscht.

Beispiel für Interface

```
interface Employee {  
    name: string;  
    salary: number;  
}
```

```
function doubleSalaryForEmployee(employee: Employee) {  
    employee.salary = employee.salary * 2;  
}
```

```
const employee2: Employee = new Employee();
```

TS2693: 'Employee' only refers to a type, but is being used as a value here.

Create class 'Employee' Alt+Umschalt+Eingabe

More actions... Alt+Eingabe

Optionale Felder

```
interface Friend {  
    firstName: string;  
    lastName?: string;  
}  
  
function getFullName(friend: Friend) {  
    let fullName = friend.firstName;  
    if(friend.lastName) {  
        fullName += " " + friend.lastName;  
    }  
    return fullName;  
}
```

Optionale Felder

- Da die Eigenschaft *lastName* optional ist, lässt sich die Funktion *getFullName* wie folgt aufrufen:

```
getFullName({firstName: "Thomas", lastName: "Huber"});  
getFullName({firstName: "Thomas"});
```

Beispiel für Klasse

```
class Employee {  
    name: string;  
    salary: number;  
  
    doubleSalaryForEmployee() {  
        this.salary = this.salary * 2;  
    }  
}
```

```
const employee: Employee = new Employee();  
employee.doubleSalaryForEmployee();
```


Aufbau einer Klasse

- Eine Klasse kann Eigenschaften, Methoden und maximal einen Konstruktor besitzen.

```
class Friend {  
    firstName: string;  
    lastName?: string;  
  
    constructor(firstName: string, lastName?: string) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

Instanz einer Klasse

- Eine Instanz einer Klasse – also ein Objekt – kann wie folgt erzeugt werden:

```
let friend1 = new Friend("Thomas", "Huber");  
let friend2 = new Friend("Julia");  
console.log(friend1.getFullName()); // Logs "Thomas Huber"  
console.log(friend2.getFullName()); // Logs "Julia"
```

Implementierung eines Interfaces

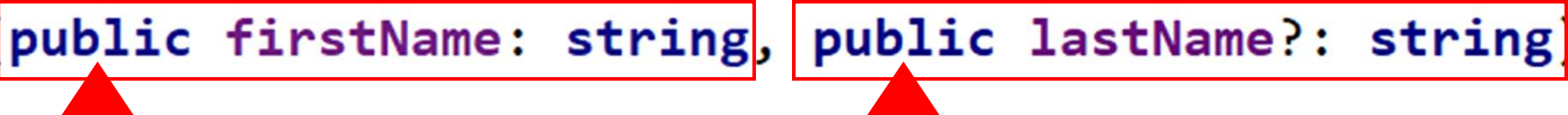
- Klassen können in TypeScript ein oder mehrere Interfaces implementieren.

```
interface Developer {  
    knowsTypeScript: boolean;  
}  
class Friend implements Developer {  
    knowsTypeScript: boolean;  
}
```

Parameter-Properties

- Parameter-Properties sind eine Kurzschreibweise in TypeScript, um Eigenschaften zu erstellen, die über einen Konstruktor-Parameter initialisiert werden.
- Durch Platzieren eines Modifiers vor einem Konstruktor-Parameter wird die Eigenschaft automatisch erstellt und mit dem Wert des Konstruktor-Parameters initialisiert.

```
class Friend {  
  constructor(public firstName: string, public lastName?: string) {}  
}
```



Statische Eigenschaften

- Unabhängig von der Anzahl der erzeugten Objekte existiert dieser Wert nur einmal pro Klasse.

```
class Friend {  
    static friendCounter: number = 0;  
    constructor() {  
        Friend.friendCounter++;  
    }  
}
```

Vererbung

- Die Klasse *Developer* erbt die Eigenschaft *firstName* der Klasse *Friend*.
- Auch Methoden werden geerbt.

```
class Friend {  
    constructor(public firstName: string) { }  
}  
class Developer extends Friend {  
    knowsTypeScript: boolean;  
}
```

Abstrakte Klassen

```
abstract class Friend {  
    constructor(public firstName: string) { }  
    abstract sayHello(): void;  
}  
  
class Developer extends Friend {  
    knowsTypeScript: boolean;  
    sayHello() {  
        console.log(`Hi, I'm ${this.firstName}`)  
    }  
}
```

Standardwerte für Parameter

```
function getFullName(firstName: string = "Julia", lastName?: string) {  
    if (lastName) {  
        return firstName + ' ' + lastName;  
    }  
    else {  
        return firstName;  
    }  
}
```


Funktionen

- Wie in JavaScript gibt es benannte und anonyme Funktionen.

```
function multiply(x, y) {  
    return x * y;  
}
```

```
let add = function(x, y) { return x + y; };
```

Funktionen als Parameter

```
function loadAdminName(callback: (adminName: string) => void) {  
    // Hier bspw. Web-API-Aufruf, welcher Namen lädt  
    callback("Thomas");  
}
```

Arrow Functions (vgl. Lambdas)

```
function loadName() {  
    this.dataLoader.loadAdminName(loadedName =>  
        this._adminName = loadedName  
    )  
}
```

Module

- Module erlauben, den TypeScript-Code in einzelne Dateien aufzuteilen.

Export eines Typs

- Um auf die Klassen eines Moduls von außen zugreifen zu können, müssen solche Klassen explizit exportiert werden.
- z.B. Datei *friends.ts*:

```
export class Friend {  
    constructor(public firstName: string,  
                public lastName: string) {  
  
    }  
}
```

Import eines Typs

- Will man anschließend in einer anderen Klasse darauf zugreifen, muss man die Klasse entsprechend importieren.

```
import { Friend } from './friends';
```

```
let friend = new Friend("Thomas", "Huber");  
console.log(friend.firstName);  
console.log(friend.lastName);
```

Export mehrerer Typen

```
class Friend {  
    constructor(public firstName: string) { }  
}  
class Developer extends Friend {  
    knowsTypeScript: boolean;  
}  
class Skateboarder extends Friend {  
    makeKickflip() {  
        console.log(this.firstName + " made a kickflip");  
    }  
}  
export { Developer, Skateboarder }
```

Import mehrerer Typen

```
import { Developer, Skateboarder } from './friends';
```


Verwendung von Typaliasen

```
export { Developer as Coder }
```

```
import { Coder } from './friends';  
let dev = new Coder("Thomas");  
dev.knowsTypeScript = true;
```

Export von Funktionen und Variablen

```
class Friend {  
    constructor(public firstName: string) { }  
}  
  
export let FRIENDS: Friend[] = [  
    new Friend("Sara"),  
    new Friend("Anna"),  
    new Friend("Thomas")];  
  
export function printFriend(friend: Friend) {  
    console.log(friend.firstName);  
}
```

Dekoratoren

- In TypeScript lassen sich Dekoratoren für folgende Ziele entwickeln:
 - Klasse
 - Property
 - Accessor
 - Methode
 - Parameter

```
@Component  
class Friend{ }
```