

# TypeScript

1 EINFÜHRUNG	3
1.1 Voraussetzung	3
1.2 Kompilieren	3
1.2.1 Target	3
1.2.2 watch	3
1.3 Playground	3
2 PROJEKT ERSTELLEN	4
2.1 npm init	4
2.2 Pakete installieren	4
2.3 Programmstruktur	5
2.3.1 index.html	6
2.3.2 index.ts	6
2.4 tsconfig.json - Konfiguration Typescript-Compiler	6
2.4.1 tsc --init	6
2.4.2 Konfiguration	6
2.5 Konfiguration Entwicklungsumgebung	7
2.5.1 Live Server	7
2.5.2 tsc	7
2.6 One-Click	8
2.6.1 Live Server als Script	8
2.6.2 Concurrently	8
2.7 Bundles	9
2.7.1 Klassen	9
2.7.2 Browserify	9
2.7.3 bundled export	10
2.7.4 Minify	10
3 LINTING	11
3.1 Packages	11
3.2 .eslintrc	11
3.2.1 Weitere Regeln	11
4 TYPISIERUNG	13
4.1 Struktur	13
4.1.1 Variable	13
4.1.2 Funktion	13
4.2 Fundamentale Typen	13
4.2.1 Default Parameters	13
4.2.2 Union Types	13
4.2.3 Type Aliases	14
4.3 Interfaces	14
4.4 Klassen	14
4.4.1 Klassen mit Konstruktor	15
4.4.2 Klassenmethoden	15
4.4.3 public/private/protected/static	16
4.4.4 Kurzvariante Konstruktor	16
4.5 Eigene Datei	17
4.6 Vererbung	17

4.7 Properties	18
4.7.1 getter	18
4.7.2 getter und setter	18
4.8 Methodenpointer	18

# 1 Einführung

TypeScript wurde von Microsoft entwickelt, um eine **Typisierung** von Javascript zu ermöglichen. Es soll also erreicht werden, dass schon zur Entwicklungszeit überprüft wird, ob **Parameter den richtigen Typ** aufweisen. TypeScript wird üblicherweise beim Build des Projekts **automatisch in eine JavaScript-Datei** übersetzt, d.h. der Browser sieht nach wie vor nur Javascript-Code.

Der Umstieg ist sehr einfach, weil jede Javascript-Datei automatisch gültiges TypeScript ist.

## 1.1 Voraussetzung

Das Node-Modul für Typescript muss installiert sein: **npm install -g typescript@latest**

```
C:\>npm install -g typescript@latest
changed 1 package, and audited 2 packages in 6s
found 0 vulnerabilities
C:\>_
```

Hinweis: in meinem Fall wurde Typescript nur aktualisiert, weil schon vorher vorhanden.

Prüfen mit **tsc -v**:

```
C:\>tsc -v
Version 4.6.4
```

## 1.2 Kompilieren

Mit obigem **tsc** kann man dann eine ts-Datei in eine js-Datei kompilieren:

```
C:\_PR\CSharp\PR4\404_Typescript_Demo>tsc index.ts
C:\_PR\CSharp\PR4\404_Typescript_Demo>
```

### 1.2.1 Target

Die Zielversion von Javascript kann man über die Option **-target** angeben, z.B. **-target es6**:

```
C:\_PR\CSharp\PR4\404_Typescript_Demo>tsc index.ts -target es6
C:\_PR\CSharp\PR4\404_Typescript_Demo>_
```

### 1.2.2 watch

Ähnlich wie für dotnet gibt es auch hier einen watch-Modus, der automatisch kompiliert, wenn sich die Datei ändert:

```
PS C:\_PR\CSharp\PR4\404_Typescript_Demo> tsc index.ts -target es5 --watch
11:06:47 - Compilation complete. Watching for file changes.

11:08:11 - File change detected. Starting incremental compilation...

11:08:12 - Compilation complete. Watching for file changes.
```

## 1.3 Playground

Es gibt mehrere Online-Plattformen, auf denen man Typescript-Code ausprobieren kann, ohne dass man ein neues Projekt aufsetzen muss, z.B. <https://playcode.io/typescript>.

## 2 Projekt erstellen

Idealerweise erstellt man mit npm ein neues Projekt, in dem alle benötigten Tools sowie das Kompilieren konfiguriert wird. Das soll an einem minimalen jQuery-Beispiel demonstriert werden.

### 2.1 npm init

Zuerst mit **npm init** ein Projekt erstellen. Dabei werden auf der Konsole einige Eingaben zum Projekt angefordert.

```
C:\_PR\CSharp\PR4\300-399\Typescript_JQuery>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (typescript_jquery)
version: (1.0.0)
description: Hello world for jQuery with Typescript
entry point: (index.js)
test command:
git repository:
keywords:
author: Robert Grueneis
license: (ISC)
About to write to C:\_PR\CSharp\PR4\300-399\Typescript_JQuery\package.json:
{
  "name": "typescript_jquery",
  "version": "1.0.0",
  "description": "Hello world for jQuery with Typescript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Robert Grueneis",
  "license": "ISC"
}

Is this OK? (yes)
C:\_PR\CSharp\PR4\300-399\Typescript_JQuery>_
```

Letztendlich entsteht die Datei **package.json**.

```
{
  "name": "typescript_jquery",
  "version": "1.0.0",
  "description": "Hello world for jQuery with Typescript",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Robert Grueneis",
  "license": "ISC"
}
```

### 2.2 Pakete installieren

Mit npm oder auch yarn dann jQuery installieren: **yarn add jquery** bzw. **npm install jquery**

```
C:\_PR\CSharp\PR4\300-399\Typescript_JQuery>yarn add jquery
yarn add v1.22.18
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

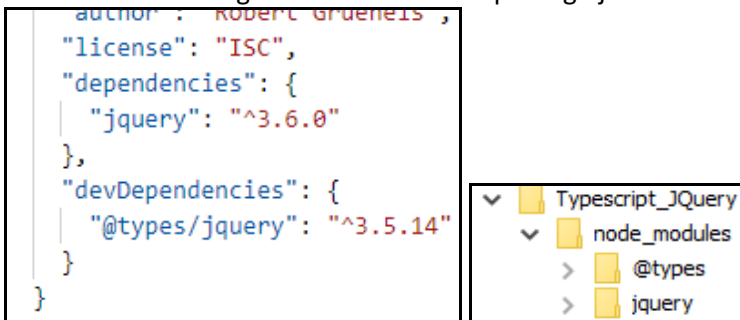
success Saved 1 new dependency.
info Direct dependencies
└─ jquery@3.6.0
info All dependencies
└─ jquery@3.6.0
Done in 0.42s.
```

Zusätzlich wie oben beschrieben noch die Type-Informationen installieren – diese werden aber nur zur Entwicklungszeit benötigt, daher: **yarn add --dev @types/jquery** bzw. **npm install--save-dev @types/jquery**

```
C:\_PR\CSharp\PR4\300-399\Typescript_JQuery>yarn add --dev @types/jquery
yarn add v1.22.18
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

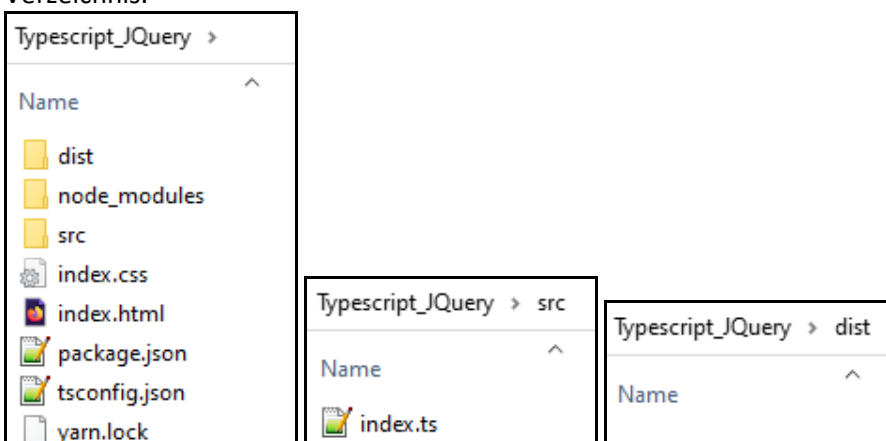
success Saved lockfile.
success Saved 2 new dependencies.
info Direct dependencies
└─ @types/jquery@3.5.14
info All dependencies
├─ @types/jquery@3.5.14
└─ @types/sizzle@2.3.3
Done in 0.99s.
```

Diese beiden Einträge sind somit auch in package.json zu finden und befinden sich im Folder node\_modules:



## 2.3 Programmstruktur

Es bietet sich an, die Typescript-Dateien in einen eigenen Ordner zu platzieren und einen zusätzlichen Ordner für den daraus generierten Javascript-Code zu erstellen. Die HTML- und CSS-Dateien bleiben üblicherweise im Root-Verzeichnis.



### 2.3.1 index.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8" />
  <title>Typescript jQuery Demo</title>
  <link rel="stylesheet" href="index.css" />
</head>

<body>
  Status:<span id="lblMsg"></span>

  <script src="node_modules/jquery/dist/jquery.js"></script>
  <script src="dist/index.js"></script>
</body>

</html>
```

### 2.3.2 index.ts

```
$(_ => {
  console.log('window loaded');
  $('#lblMsg').html('Ok');
});
```

## 2.4 tsconfig.json - Konfiguration Typescript-Compiler

Die Konfiguration des Typescript-Compilers erfolgt über die Datei **tsconfig.json**. Dort kann man sehr viele Optionen einstellen.

### 2.4.1 tsc --init

Am besten erzeugt man mit **tsc --init** eine Datei, bei der alle Optionen aufgelistet aber auskommentiert sind. Dann ändert man und aktiviert man jene Einstellungen, die wesentlich sind.

```
C:\_PR\CSharp\PR4\300-399\Typescript_JQuery>tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true

You can learn more at https://aka.ms/tsconfig.json
```

### 2.4.2 Konfiguration

Folgende Einstellungen sollten geändert werden.

Option	Wert	Beschreibung
target	es5	Auf diese Javascript-Version wird kompiliert.
moduleResolution	node	Wichtig wenn man Module im Browser verwenden will
outDir	./dist	Wohin werden die Javascript-Dateien erzeugt?
sourceRoot	./src	Für Debugger: Wo stehen die Source-Files?

inlineSourceMap	true	Wird aufgrund der Option sourceRoot benötigt
-----------------	------	--

Außerdem sollte man angeben, welche Files kompiliert werden sollen. Das erfolgt über die Optionen **include** und **exclude** (diese sind Geschwisterknoten zu compilerOptions):

```
{
  "compilerOptions": { ...
  },
  "include": [
    |  "./src/*.ts"
  ],
  "exclude": [
    |  "node_modules",
    |  "**/node_modules/**"
  ]
}
```

## 2.5 Konfiguration Entwicklungsumgebung

Idealerweise soll im Browser immer die aktuelle Version der App angezeigt werden, d.h. bei Änderung in HTML bzw. Code soll ein Reload erfolgen

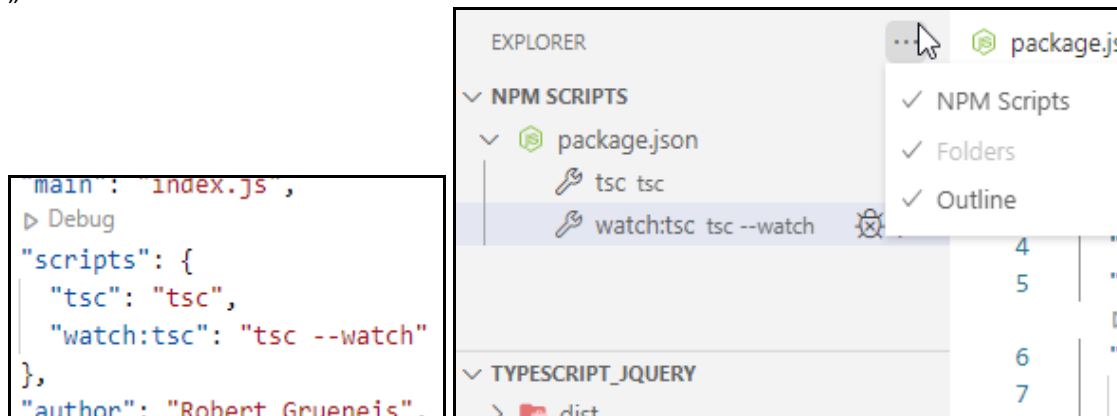
### 2.5.1 Live Server

Der erste Teil wird durch das Plugin Live Server abgedeckt. Also index.html auswählen und „Go Live“ klicken.

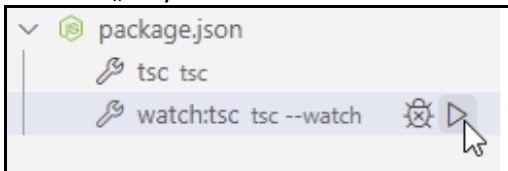


### 2.5.2 tsc

Der Compiler wird am besten über Scripts in package.json konfiguriert. Üblicherweise werden zwei Befehle konfiguriert – einer für das manuelle Erstellen und einer für den Watch-Modus. Diese scheinen dann im Tab „NPM SCRIPTS“ auf:



Mit dem „Play“-Button kann dann der Watch-Modus gestartet werden:



Es wird ein neues Terminal geöffnet. Dort sieht man die Ausgaben und kann man über den „Papierkorb“-Button den Task auch wieder beenden:

```

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
[10:49:59] File change detected. Starting incremental compilation...
[10:49:59] Found 0 errors. Watching for file changes.

```

## 2.6 One-Click

Zu guter Letzt soll das Projekt noch so konfiguriert werden, dass man mit einem Klick die App starten kann und sich diese bei allen Änderungen aktualisiert.

### 2.6.1 Live Server als Script

Zuerst den Live-Server global installieren:

```

C:\>npm install -g live-server
npm WARN deprecated source-map-url@0.4.1: See https://github.com/lydell/source-map-url#deprecated
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
npm WARN deprecated source-map-resolve@0.5.3: See https://github.com/lydell/source-map-resolve#deprecated
npm WARN deprecated chokidar@2.1.8: Chokidar 2 does not receive security updates since 2019. Upgrade to chokidar@3.4.0: Please upgrade to version 7 or higher. Older versions may use Math.random()
npm WARN deprecated opn@6.0.0: The package has been renamed to `open`

added 204 packages, and audited 205 packages in 2m

3 high severity vulnerabilities

To address all issues, run:
  npm audit fix

Run `npm audit` for details.

```

Falls man in PowerShell ein Problem hat, hilft folgender Befehl:

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted
```

Dann einen Eintrag in package.json erstellen:

```

"scripts": {
  "tsc": "tsc",
  "watch:tsc": "tsc --watch",
  "watch:app": "live-server"
},

```

### 2.6.2 Concurrently

Um mehrere Scripts gleichzeitig ausführen zu können muss man das Tool Concurrently installieren: **yarn add -dev concurrently**

Damit kann man dann mehrere Tasks zusammenfassen:

```

"scripts": {
  "tsc": "tsc",
  "watch:tsc": "tsc --watch",
  "watch:app": "live-server",
  "watch:all": "concurrently npm:watch:tsc npm:watch:app"
},

```

Startet man jetzt den Task watch:all hat man den gewünschten Effekt: bei jeder Änderung in HTML, CSS oder Typescript wird die Anzeige im Browser aktualisiert.



## 2.7 Bundles

Will man seinen Code auf mehrere Files/Module verteilen, muss man das Laden der Module für den Browser vorbereiten.

### 2.7.1 Klassen

Als Beispiel sollen zwei Klassen Person und davon abgeleitet Student erstellt werden (Details zur Typescript-Syntax weiter unten):

**src\person.ts**

```
export class Person {
  constructor(protected firstname: string, protected lastname: string, protected age: number) { }
  toString(): string {
    return `${this.firstname} ${this.lastname} (${this.age})`;
  }
}
```

**src\student.ts**

```
import { Person } from './person';

export class Student extends Person {
  constructor(firstname: string, lastname: string, age: number, private school: string) {
    super(firstname, lastname, age);
  }
  toString(): string {
    return `${super.toString()} from ${this.school}`;
  }
}
```

**src\index.ts**

```
import { Person } from './person';
import { Student } from './student';

$( _ => {
  console.log('document ready');

  const p = new Person('Hansi', 'Huber', 66);
  console.log(p.toString());

  const student: Student = new Student('Fritzi', 'Mueller', 16, 'HTL-Grieskirchen');
  console.log(`Student: ${student.toString()}`);
});
```

### 2.7.2 Browserify

Ein möglicher Bundler ist **Browserify** (<https://www.npmjs.com/package/browserify>) bzw. **Watchify** (<https://www.npmjs.com/package/watchify>).

Diese beiden Tools daher entweder global oder lokal installieren:

```
npm install -g browserify / npm install --save-dev browserify / yarn add --dev browserify
npm install -g watchify / npm install --save-dev watchify / yarn add --dev watchify
```

Das Bundle dann in einen eigenen Folder generieren, z.B. bundle.

Folgende Tasks daher in package.json bei scripts notieren:

```
"bundle": "browserify dist/index.js dist/person.js dist/student.js -o bundle/app.js",
"watch:bundle": "watchify dist/index.js dist/person.js dist/student.js -o bundle/app.js",
"watch:all": "concurrently npm:watch:tsc npm:watch:app npm:watch:bundle"
```

Und dieses Bundle dann in index.html verwenden:

```
<body>
  ...

  <script src="node_modules/jquery/dist/jquery.js"></script>
  <script src="node_modules/bootstrap/dist/js/bootstrap.js"></script>

  <script src="bundle/app.js"></script>
</body>
```

### 2.7.3 bundled export

Damit man nicht alle Dateien separat im npm-Script angeben muss, werden oft die export-Anweisungen in einer eigenen Datei gesammelt.

```
models.ts
src > models.ts
1  export * from './person';
2  export * from './student';
```

Diese Datei kann dann einerseits als Referenz bei der Verwendung in index.ts herangezogen werden:

```
import { Person, Student } from './models';
// import { Person } from './person';
// import { Student } from './student';
```

Andererseits muss nur noch diese Datei beim Bundler angegeben werden:

```
"bundle": "browserify dist/index.js dist/models.js -o bundle/app.js",
"watch:bundle": "watchify dist/index.js dist/models.js -o bundle/app.js",
```

### 2.7.4 Minify

Aus der entstandenen Datei könnte man dann noch eine .min-Version erzeugen. Dazu gibt es mehrere Node-Module, z.B. Terser (<https://github.com/terser/terser>).

Installieren wie gewohnt mit `npm install terser` bzw. `yarn add terser`

Und damit einen neuen Task in package.json erstellen:

```
"minify:app": "terser bundle/app.js -o bundle/app.min.js -c -m"
```

## 3 Linting

Lintint funktioniert wie bei Javascript mit ESLint. Bis vor kurzem gab es ein separates Plugin TSLint für Typescript – das ist aber „deprecated“ → siehe [https://dev.to/sam\\_piggott/setting-up-typescript-with-eslint-prettier-for-visual-studio-code-1e3h](https://dev.to/sam_piggott/setting-up-typescript-with-eslint-prettier-for-visual-studio-code-1e3h)

Auch muss das globale Config-File für eslint in settings.json deaktiviert werden!

### 3.1 Packages

Mit npm oder yarn die Pakete lokal installieren:

- **typescript**
- **eslint**
- **@typescript-eslint/eslint-plugin** - A plugin for ESLint to support TypeScript specifically
- **@typescript-eslint/parser** - Further support for ESLint to lint TypeScript source files

Also:

```
yarn add eslint typescript typescript-eslint/eslint-plugin @typescript-eslint/parser --dev
```

Diese werden damit in package.json bei devDependencies eingetragen:

```
"devDependencies": {
  "@types/jquery": "^3.5.14",
  "@typescript-eslint/eslint-plugin": "^5.22.0",
  "@typescript-eslint/parser": "^5.22.0",
  "concurrently": "^7.1.0",
  "eslint": "^8.14.0",
  "typescript": "^4.6.4"
}
```

### 3.2 .eslintrc

Dann im root-Verzeichnis des Projekts die Datei **.eslintrc** mit folgendem Inhalt erzeugen:

```
{
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended"
  ],
  "globals": {
    "$": "readonly"
  },
  "rules": {
    "quotes": [2, "single"]
  }
}
```

Danach Visual Studio schließen und neu starten.

#### 3.2.1 Weitere Regeln

Zusätzliche Regeln kann man dann z.B. so eintragen:

```
"rules": {  
  "quotes": [2, "single"],  
  "@typescript-eslint/no-unused-vars": [2, {  
    "varsIgnorePattern": "^_",  
    "argsIgnorePattern": "^_"  
  }]  
}
```

## 4 Typisierung

Um TypeScript richtig verwenden zu können muss man natürlich wissen, welche Typen es gibt bzw. wo und wie man diese angibt.

Dazu siehe auch: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html>

### 4.1 Struktur

#### 4.1.1 Variable

Bei der Definition einer Variablen eines bestimmten Typs gibt man den Typ hinter dem Namen durch **Doppelpunkt** getrennt an:

```
var age: number = 66;
```

#### 4.1.2 Funktion

Wie bei den Variablen wird der Typ der Parameter hinter dem Namen angegeben, der Returntyp wird zwischen Parameterliste und Funktionsbody geschrieben:

```
function showFullName(name: string, age: number): string {
  return `${name} (${age})`;
}
```

### 4.2 Fundamentale Typen

Die Grundtypen, die man sich erwarten würde, gibt es auch und sie heißen auch so wie in anderen Sprachen.

C#	TypeScript
string	string
int, float, ...	number
bool	boolean
List<string>	string[] oder Array<string>
List<int>	number[] oder Array<number>
enum Gender { Male, Female };	enum Gender { Male, Female };
object	any
void	void

#### 4.2.1 Default Parameters

Parameter am Ende einer Liste können als optional deklariert werden, indem man ein **?** hinter den Parameternamen (also nicht dem Typ!) schreibt.

```
function callMe(paraA: string, paraB?: string) {
  let s = paraA;
  if (paraB) s += `_${paraB}`;
  return s;
}
```

Man kann auch Defaultwerte definieren:

```
private static timeStamp(includeDate: boolean = false, includeMillis: boolean = true): string {
  var now = new Date();
}
```

#### 4.2.2 Union Types

Mit dem Pipe-Symbol **|** kann man einen sogenannten Union Type angeben. Das sind Typen, die einem der angegebenen Type entsprechen.

```
function printId(id: number | string) {
  console.log(`Your Id is: ${id}`);
}
printId(666); // Ok
printId('777'); // Ok
// printId({ id: 888 }); // Not Ok
```

### 4.2.3 Type Aliases

Mit dem Schlüsselwort **type** kann man neue Namen für Objekt-Typen oder Union Types definieren.

```
type Point = { x: number; y: number; };
function printCoord(pt: Point) {
  console.log(`The coordinate's x value is ${pt.x}`);
  console.log(`The coordinate's y value is ${pt.y}`);
}
printCoord({ x: 100, y: 100 });
```

```
function printID(id: ID) {
  console.log(`Your Id is: ${id}`);
}
type ID = number | string;
const idA: ID = 66;
const idB: ID = '77';
printID(idA);
printID(idB);
printID(6);
printID('7');
```

### 4.3 Interfaces

Wie in anderen Programmiersprachen auch definieren Interfaces nur eine Struktur. Es wird also sichergestellt, dass das übergebene Objekt die **angeführten Properties** besitzt. Das Objekt kann darüber hinaus aber auch noch andere Properties haben.

```
interface MyProps {
  propA: string;
  propB: number;
}

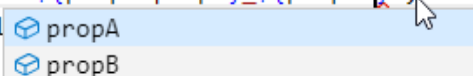
function useProps(props: MyProps): void {
  const s = `${props.propA}_${props.propB}`;
  console.log(`Using Props: ${s}`);
}

const ppp = { propA: 'aaa', propB: 123, propC: 'xyz' };
useProps(ppp);
```

Hier sind also **propA** und **propB** verpflichtend beim Aufruf von **useProps**. Dass das Objekt **ppp** noch die Property **propC** hat, ist irrelevant.

Intellisense funktioniert dabei wie erwartet:

```
function useProps(props: MyProps): void {
  const s = `${props.propA}_${props.}`;
  console.log(s);
}
```



### 4.4 Klassen

Zu den in TypeScript bekannten Typen kann man auch eigene Klassen definieren:

```
class Person {
  firstname: string;
  lastname: string;
}
```

Die Verwendung erfolgt dann, wie man es sich erwarten würde.

```
function getName(person: Person) {
    return `${person.firstname} ${person.lastname}`;
}
let person = { firstname: 'Hansi', lastname: 'Huber' };
let fullname = getName(person);
```

Wie man sieht, werden nur die Properties zugewiesener Objekte überprüft („duck typing“).

Auch dabei werden Fehler wieder zur Compilezeit erkannt:

```
let person = {firstname:'Hansi',xxx:'Huber'};
let fullname = getName(person);
```

	Code	Description	Project	File
✖	TS2345	(TS) Argument of type '{ firstname: string; xxx: string; }' is not assignable to parameter of type 'Person'. Property 'lastname' is missing in type '{ firstname: string; xxx: string; }'.	Scripts (tsconfig...	tester1.ts

#### 4.4.1 Klassen mit Konstruktor

Ähnlich wie in C# oder Java kann man auch typisierte Konstruktoren definieren. Dieser heißt in TypeScript immer **constructor**, und nicht wie die Klasse:

```
class Person {
    firstname: string;
    lastname: string;
    age: number;
    constructor(firstname: string, lastname: string, age: number) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }
}
```

```
function getName(person: Person) {
    return `${person.firstname} ${person.lastname} (${person.age})`;
}
let person = new Person('Hansi', 'Huber', 66);
let fullname = getName(person);
```

Aber auch hier wird vor dem Senden zum Browser wieder eine normale Javascript-Datei erzeugt:

```
var Person = (function () {
    function Person(firstname, lastname, age) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }
    return Person;
})();
function getName(person) {
    return person.firstname + " " + person.lastname + " (" + person.age + ")";
}
var person = new Person('Hansi', 'Huber', 66);
var fullname = getName(person);
```

#### 4.4.2 Klassenmethoden

Auch Klassen werden ähnlich wie in anderen Programmiersprachen definiert, indem man Properties und Methoden typisiert angibt.

```

class Person {
  firstname: string;
  lastname: string;
  age: number;
  constructor(firstname: string, lastname: string, age: number) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
  }
  toString(): string {
    return `${this.firstname} ${this.lastname} (${this.age})`;
  }
}

const p = new Person('Hansi', 'Huber', 66);
console.log(p.toString());

```

Für den Browser entsteht dabei natürlich ein normales JavaScript-Objekt mit Prototyp:

```

var Person = (function () {
  function Person(firstname, lastname, age) {
    this.firstname = firstname;
    this.lastname = lastname;
    this.age = age;
  }
  Person.prototype.toString = function () {
    return this.firstname + " " + this.lastname + " (" + this.age + ")";
  };
  return Person;
})();

```

Man muss bei der Verwendung aber immer **this** angeben und nicht weglassen, wie in C# oder Java!

#### 4.4.3 public/private/protected/static

In TypeScript sind per **Default** alle Properties/Methoden public. Man könnte sie aber auch explizit mit **public** kennzeichnen oder eben auch **private** oder **protected** deklarieren.

Auch das Schlüsselwort **static** funktioniert so wie in C# bzw. Java.

#### 4.4.4 Kurzvariante Konstruktor

Für Konstruktoren gibt es noch die Kurzvariante, dass man keine Felder direkt notiert, sondern nur die Parameter im Konstruktor angibt, dabei aber mit **protected/private/public** notiert:

```

class Person {
  constructor(protected firstname: string, protected lastname: string, protected age: number) { }
  toString(): string {
    return `${this.firstname} ${this.lastname} (${this.age})`;
  }
}

```

Es wird trotzdem praktisch das gleiche Javascript erzeugt:



```

var Person = (function () {
    function Person(firstname, lastname, age) {
        this.firstname = firstname;
        this.lastname = lastname;
        this.age = age;
    }
    Person.prototype.toString = function () {
        return this.firstname + " " + this.lastname + " (" + this.age + ")";
    };
    return Person;
})();

```

#### 4.5 Eigene Datei

Wie auch in C# und Java üblich sollten Klassen in eigenen Dateien definiert werden. Damit die Klasse aber auch verwendet werden kann, muss sie mit **export class** notiert werden:

```

export class Person {
    constructor(protected firstname: string, protected lastname: string, protected age: number) { }
    toString(): string {
        return this.firstname + " " + this.lastname + " (" + this.age + ")";
    }
}

```

Bei der Verwendung muss die Datei dann mit import

```
const p = new Person('Hansi', 'Huber', 66);
```

Import 'Person' from module "../data/person"

Add all missing imports

```
import { Person } from '../data/person';
```

#### 4.6 Vererbung

Die Ableitung von Klassen ist syntaktisch an Java angelehnt – man braucht das Schlüsselwort **extends**.

```

class Student extends Person {
    constructor(firstname: string, lastname: string, age: number, private school: string) {
        super(firstname, lastname, age);
    }
    toString(): string {
        const s = `${this.firstname} ${this.lastname} from ${this.school}`;
        return s;
    }
}

```

**Achtung:** damit man auf Variablen der Basisklasse zugreifen kann, müssen diese **protected** (bzw. **public**) notiert werden.

```

class Person {
    constructor(protected firstname: string, protected lastname: string, protected age: number) { }
    toString(): string {
        return this.firstname + " " + this.lastname + " (" + this.age + ")";
    }
}

```

Natürlich kann man auch hier wieder die Implementierung der Basisklasse benutzen:

```

class Student extends Person {
    constructor(firstname: string, lastname: string, age: number, private school: string) {
        super(firstname, lastname, age);
    }
    toString(): string {
        const s = `${super.toString()} from ${this.school}`;
        return s;
    }
}

```

Auch hier zum Vergleich der entstandene Javascript-Code:

```

var Student = (function (_super) {
    __extends(Student, _super);
    function Student(firstname, lastname, age, school) {
        _super.call(this, firstname, lastname, age);
        this.school = school;
    }
    Student.prototype.toString = function () {
        //const s = `${this.firstname} ${this.lastname} from ${this.school}`;
        var s = _super.prototype.toString.call(this) + " from " + this.school;
        return s;
    };
    return Student;
})(Person));

```

## 4.7 Properties

Wie von C# gewohnt sind Properties Funktionen, die wie eine Instanzvariable anzusprechen sind.

### 4.7.1 getter

Eine Property für Name könnte dann so aussehen:

```

get name(): string {
    return `${this.firstname} ${this.lastname}`;
}

```

### 4.7.2 getter und setter

Auch das ist keine Überraschung:

```

private _email: string;
get email(): string {
    return this._email;
}
set email(value: string) {
    if (!value) throw new Error('Please supply a valid email');
    this._email = value;
}

```

Hinweise:

- Es ist oft üblich, die Instanzvariable mit einem „\_“ beginnen zu lassen. Dann kann man die Property praktisch gleich wie die Variable benennen.
- Die Unterwellung=Warning kommt daher:

```

Declaration of instance field not allowed after declaration of
instance method. Instead, this should come at the beginning of
the class/interface. (member-ordering) tslint(1)

```

Peek Problem Quick Fix...

```

private _email: string;

```

Das könnte man in tslint.json einstellen (siehe unten)

## 4.8 Methodenpointer

In Typescript gibt es auch die Möglichkeit, Funktionssignaturen festzulegen.

Die allgemeine Struktur lautet:

**(p1: type1, p2: type2) => returnType**

Beispiel:

```
function workWithCallback(nr: number, fct: (idx: number) => void) {  
  for (let i = 0; i < nr; i++) {  
    fct(i);  
  }  
}
```

Die Verwendung erfolgt dann meist in Form eines Lambda-Ausdrucks:

```
workWithCallback(5, x => console.log(`Work with index ${x}`));
```