**Kostenloses eBook**

# LERNEN TypeScript

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#typescript

# Inhaltsverzeichnis

# Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: typescript

It is an unofficial and free TypeScript ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official TypeScript.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Kapitel 1: Erste Schritte mit TypeScript

## Bemerkungen

TypeScript soll eine Obermenge von JavaScript sein, die in JavaScript übersetzt wird. Durch die Erzeugung von ECMAScript-kompatiblem Code kann TypeScript neue Sprachfunktionen einführen und dabei die Kompatibilität mit vorhandenen JavaScript-Engines beibehalten. ES3, ES5 und ES6 sind derzeit unterstützte Ziele.

Optionale Typen sind ein Hauptmerkmal. Typen ermöglichen eine statische Überprüfung mit dem Ziel, Fehler frühzeitig zu finden, und können die Funktionsweise mit Funktionen wie Code-Refactoring verbessern.

TypeScript ist eine von Microsoft entwickelte Open Source- und plattformübergreifende Programmiersprache. Der Quellcode ist auf GitHub verfügbar .

## Versionen

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 2.4.1 | 2017-06-27 |
| 2.3.2 | 2017-04-28 |
| 2.3.1 | 2017-04-25 |
| 2.3.0 Beta | 2017-04-04 |
| 2.2.2 | 2017-03-13 |
| 2.2 | 2017-02-17 |
| 2.1.6 | 2017-02-07 |
| 2.2 Beta | 2017-02-02 |
| 2.1.5 | 2017-01-05 |
| 2.1.4 | 2016-12-05 |
| 2.0.8 | 2016-11-08 |
| 2.0.7 | 2016-11-03 |
| 2.0.6 | 2016-10-23 |
| 2.0.5 | 2016-09-22 |

| Ausführung | Veröffentlichungsdatum |
|---|---|
| 2.0 Beta | 2016-07-08 |
| 1.8.10 | 2016-04-09 |
| 1.8.9 | 2016-03-16 |
| 1.8.5 | 2016-03-02 |
| 1.8.2 | 2016-02-17 |
| 1.7.5 | 2015-12-14 |
| 1.7 | 2015-11-20 |
| 1.6 | 2015-09-11 |
| 1.5.4 | 2015-07-15 |
| 1,5 | 2015-07-15 |
| 1.4 | 2015-01-13 |
| 1.3 | 2014-10-28 |
| 1.1.0.1 | 2014-09-23 |

## Examples

### Installation und Einrichtung

# Hintergrund

TypeScript ist eine typisierte JavaScript-Obermenge, die direkt in JavaScript-Code kompiliert wird. TypScript-Dateien verwenden im Allgemeinen die Erweiterung `.ts` . Viele IDEs unterstützen TypeScript, ohne dass ein anderes Setup erforderlich ist, aber TypeScript kann auch mit dem TypeScript Node.JS-Paket von der Befehlszeile aus kompiliert werden.

# IDEs

## Visual Studio

- `Visual Studio 2015` enthält TypeScript.
- `Visual Studio 2013 Update 2` oder höher enthält TypeScript oder Sie können TypeScript für frühere Versionen herunterladen .

## Visual Studio Code

- Visual Studio Code (vscode) bietet kontextabhängige Autovervollständigung sowie Refactoring- und Debugging-Tools für TypeScript. vscode ist selbst in TypeScript implementiert. Verfügbar für Mac OS X, Windows und Linux.

## WebStorm

- `WebStorm 2016.2` TypeScript und einen integrierten Compiler. [Webstorm ist nicht kostenlos]

## IntelliJ IDEA

- `IntelliJ IDEA 2016.2` bietet Unterstützung für Typescript und einen Compiler über ein Plugin , das vom Jetbrains-Team verwaltet wird. [IntelliJ ist nicht frei]

## Atom & Atom-Typoskript

- `Atom` unterstützt TypeScript mit dem Atom-Typoscript- Paket.

## Erhabener Text

- `Sublime Text` unterstützt TypeScript mit dem Typoscript- Paket.

# Befehlszeilenschnittstelle installieren

## Installieren Sie Node.js

## Installieren Sie das npm-Paket global

Sie können TypeScript global installieren, um von jedem Verzeichnis aus darauf zugreifen zu können.

```
npm install -g typescript
```

*oder*

## Installieren Sie das npm-Paket lokal

Sie können TypeScript lokal installieren und in package.json speichern, um auf ein Verzeichnis zu beschränken.

```
npm install typescript --save-dev
```

### Installationskanäle

Sie können installieren von:

- Stabiler Kanal: `npm install typescript`
- Betakanal: `npm install typescript@beta`
- Dev channel: `npm install typescript@next`

## TypeScript-Code wird kompiliert

Der `tsc` Kompilierungsbefehl wird mit `typescript` `tsc` , mit dem Code kompiliert werden kann.

```
tsc my-code.ts
```

Dadurch wird eine Datei `my-code.js` erstellt.

## Kompilieren Sie mit tsconfig.json

Sie können auch Kompilierungsoptionen bereitstellen, die mit Ihrem Code über eine Datei `tsconfig.json` werden. Um ein neues TypeScript-Projekt zu starten, `cd` in einem Terminalfenster in das Stammverzeichnis des `tsc --init` und `tsc --init` . Dieser Befehl generiert eine Datei " `tsconfig.json` " mit minimalen Konfigurationsoptionen (ähnlich wie unten).

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false,
        "pretty": true
    },
    "exclude": [
        "node_modules"
    ]
}
```

Mit einer Datei `tsconfig.json` , die sich im Stammverzeichnis Ihres TypeScript-Projekts befindet, können Sie den Befehl `tsc` zum Ausführen der Kompilierung verwenden.

### Hallo Welt

```
class Greeter {
    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }
    greet(): string {
        return this.greeting;
    }
```

```
};

let greeter = new Greeter("Hello, world!");
console.log(greeter.greet());
```

Hier haben wir eine Klasse, `Greeter` , die einen `constructor` und eine `greet` Methode hat. Wir können eine Instanz der Klasse mit dem `new` Schlüsselwort erstellen und eine Zeichenfolge übergeben, die die `greet` Methode an die Konsole ausgeben soll. Die Instanz unserer `Greeter` Klasse wird in der `greeter` Variablen gespeichert, die wir dann als `greet` Methode aufrufen.

### Grundlegende Syntax

TypeScript ist eine typisierte Obermenge von JavaScript, was bedeutet, dass der gesamte JavaScript-Code gültiger TypeScript-Code ist. TypeScript fügt darüber hinaus viele neue Funktionen hinzu.

Mit TypeScript wird JavaScript mehr wie eine stark typisierte, objektorientierte Sprache, die C # und Java ähnelt. Dies bedeutet, dass TypeScript-Code für große Projekte leichter zu verwenden ist und dass Code leichter zu verstehen und zu verwalten ist. Die starke Typisierung bedeutet auch, dass die Sprache vorkompiliert werden kann (und ist) und dass Variablen keine Werte zugewiesen werden können, die außerhalb ihres angegebenen Bereichs liegen. Wenn zum Beispiel eine TypeScript-Variable als Zahl deklariert ist, können Sie ihr keinen Textwert zuweisen.

Diese starke Typisierung und Objektorientierung erleichtert das Debuggen und Verwalten von TypeScript. Dies waren zwei der schwächsten Punkte von Standard-JavaScript.

## Typanmeldungen

Sie können Typdeklarationen zu Variablen, Funktionsparametern und Funktionsrückgabetypen hinzufügen. Der Typ wird wie folgt nach einem Doppelpunkt hinter dem Variablennamen geschrieben: `var num: number = 5;` Der Compiler überprüft dann die Typen (wenn möglich) während des Kompilierens und meldet Typfehler.

```
var num: number = 5;
num = "this is a string";  // error: Type 'string' is not assignable to type 'number'.
```

Die Grundtypen sind:

- `number` (sowohl Ganzzahlen als auch Gleitkommazahlen)
- `string`
- `boolean`
- `Array` Sie können die Typen der Elemente eines Arrays angeben. Es gibt zwei gleichwertige Möglichkeiten, Array-Typen zu definieren: `Array<T>` und `T[]` . Zum Beispiel:
    - `number[]` - Zahlenfeld
    - `Array<string>` - Array von Zeichenfolgen
- Tuples. Tupel haben eine feste Anzahl von Elementen mit bestimmten Typen.
    - `[boolean, string]` - Tupel, wobei das erste Element ein Boolean und das zweite Element ein String ist.

    - `[number, number, number]` - Tupel aus drei Zahlen.
- `{}` - Objekt, Sie können seine Eigenschaften oder den Indexer definieren
    - `{name: string, age: number}` - Objekt mit Namens- und Altersattributen
    - `{[key: string]: number}` - ein Wörterbuch mit Zahlen, die durch einen String indiziert sind
- `enum` - `{ Red = 0, Blue, Green }` - Aufzählung, die Zahlen zugeordnet ist
- Funktion. Sie geben Typen für die Parameter und den Rückgabewert an:
    - `(param: number) => string` - Funktion, bei der ein Parameter für die Anzahl von Zeichenfolgen zurückgegeben wird
    - `() => number` - Funktion ohne Parameter, die eine Nummer zurückgeben.
    - `(a: string, b?: boolean) => void` - Funktion, die einen String und optional einen Boolean ohne Rückgabewert übernimmt.
- `any` - Erlaubt jeden Typ. Ausdrücke, die `any` werden nicht typgeprüft.
- `void` - steht für "nothing", kann als Funktionsrückgabewert verwendet werden. Nur `null` und `undefined` sind Teil des `void` Typs.
- `never`
    - `let foo: never;` -Der Typ der Variablen unter Typwächtern, die niemals wahr sind.
    - `function error(message: string): never { throw new Error(message); }` - Als Rückgabetyp von Funktionen, die niemals zurückgegeben werden.

- `null` - Typ für den Wert `null` . `null` ist implizit Teil jedes Typs, sofern keine strengen Nullprüfungen aktiviert sind.

## Casting

Sie können das explizite Gießen über spitze Klammern durchführen, zum Beispiel:

```
var derived: MyInterface;
(<ImplementingClass>derived).someSpecificMethod();
```

Dieses Beispiel zeigt eine `derived` Klasse, die vom Compiler als `MyInterface` behandelt wird. Ohne das Casting in der zweiten Zeile würde der Compiler eine Ausnahme `someSpecificMethod()` , da `someSpecificMethod()` nicht verstanden wird. `someSpecificMethod()` Casting durch `<ImplementingClass>derived` den Compiler an, was zu tun ist.

Eine andere Art des Castings in Typescript ist das Schlüsselwort `as` :

```
var derived: MyInterface;
(derived as ImplementingClass).someSpecificMethod();
```

Seit Typescript 1.6 wird standardmäßig das Schlüsselwort `as` , da die Verwendung von `<>` in **.jsx-**Dateien mehrdeutig ist. Dies ist in der offiziellen Dokumentation zu TypeScript erwähnt .

## Klassen

Klassen können im TypeScript-Code definiert und verwendet werden. Weitere Informationen zu Klassen finden Sie auf der Dokumentationsseite Klassen .

## TypeScript REPL in Node.js

Für die Verwendung von TypeScript REPL in Node.js können Sie das Tsun-Paket verwenden

Installieren Sie es global mit

```
npm install -g tsun
```

und führen Sie Ihr Terminal oder die Eingabeaufforderung mit dem Befehl `tsun`

Anwendungsbeispiel:

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl
$ function multiply(x, y) {
..return x * y;
..}
undefined
$ multiply(3, 4)
12
```

## Ausführen von TypeScript mit ts-node

ts-node ist ein npm-Paket, mit dem der Benutzer `tsc` Dateien direkt ausführen kann, ohne dass er mit `tsc` vorkompiliert werden `tsc` . Es bietet auch REPL .

Installieren Sie ts-node global mit

```
npm install -g ts-node
```

ts-node enthält keinen TypScript-Compiler, daher müssen Sie ihn möglicherweise installieren.

```
npm install -g typescript
```

### Skript ausführen

Um ein Skript mit dem Namen *main.ts auszuführen* , führen Sie *Folgendes* aus

```
ts-node main.ts
```

```
// main.ts
console.log("Hello world");
```

Verwendungsbeispiel

```
$ ts-node main.ts
Hello world
```

### REPL ausführen

Um REPL auszuführen, führen Sie den Befehl `ts-node`

Verwendungsbeispiel

```
$ ts-node
> const sum = (a, b): number => a + b;
undefined
> sum(2, 2)
4
> .exit
```

Um REPL zu `.exit` verwenden Sie den Befehl `.exit` oder drücken Sie zweimal `.exit CTRL+C`

Erste Schritte mit TypeScript online lesen: https://riptutorial.com/de/typescript/topic/764/erste-schritte-mit-typescript

# Kapitel 2: Arrays

## Examples

### Objekt im Array suchen

# Find () verwenden

```
const inventory = [
    {name: 'apples', quantity: 2},
    {name: 'bananas', quantity: 0},
    {name: 'cherries', quantity: 5}
];

function findCherries(fruit) {
    return fruit.name === 'cherries';
}

inventory.find(findCherries); // { name: 'cherries', quantity: 5 }

/* OR */

inventory.find(e => e.name === 'apples'); // { name: 'apples', quantity: 2 }
```

Arrays online lesen: https://riptutorial.com/de/typescript/topic/9562/arrays

# Kapitel 3: Aufzählungen

## Examples

### So erhalten Sie alle Aufzählungswerte

```
enum SomeEnum { A, B }

let enumValues:Array<string>= [];

for(let value in SomeEnum) {
    if(typeof SomeEnum[value] === 'number') {
        enumValues.push(value);
    }
}

enumValues.forEach(v=> console.log(v))
//A
//B
```

### Aufzählungen mit expliziten Werten

Standardmäßig werden alle `enum` in Zahlen aufgelöst. Sagen wir mal, ob du sowas hast

```
enum MimeType {
  JPEG,
  PNG,
  PDF
}
```

Der tatsächliche Wert hinter zB `MimeType.PDF` ist `2` .

Manchmal ist es jedoch wichtig, dass das Enum auf einen anderen Typ aufgelöst wird. Beispielsweise erhalten Sie den Wert vom Backend / Frontend / einem anderen System, das definitiv eine Zeichenfolge ist. Das könnte schmerzhaft sein, aber zum Glück gibt es diese Methode:

```
enum MimeType {
  JPEG = <any>'image/jpeg',
  PNG = <any>'image/png',
  PDF = <any>'application/pdf'
}
```

Dies löst das `MimeType.PDF` in `application/pdf` .

Seit TypeScript 2.4 können Sie String-Enummen deklarieren:

```
enum MimeType {
  JPEG = 'image/jpeg',
  PNG = 'image/png',
```

```
  PDF = 'application/pdf',
}
```

Sie können numerische Werte explizit mit derselben Methode angeben

```
enum MyType {
   Value = 3,
   ValueEx = 30,
   ValueEx2 = 300
}
```

Fancier-Typen funktionieren auch, da Nicht-Konstanten-Enums beispielsweise zur Laufzeit echte Objekte sind

```
enum FancyType {
   OneArr = <any>[1],
   TwoArr = <any>[2, 2],
   ThreeArr = <any>[3, 3, 3]
}
```

wird

```
var FancyType;
(function (FancyType) {
    FancyType[FancyType["OneArr"] = [1]] = "OneArr";
    FancyType[FancyType["TwoArr"] = [2, 2]] = "TwoArr";
    FancyType[FancyType["ThreeArr"] = [3, 3, 3]] = "ThreeArr";
})(FancyType || (FancyType = {}));
```

## Benutzerdefinierte Enum-Implementierung: wird für Enums erweitert

Manchmal ist es erforderlich, Enum selbst zu implementieren. Es gibt zB keine klare Möglichkeit, andere Aufzählungen zu erweitern. Benutzerdefinierte Implementierung ermöglicht dies:

```
class Enum {
  constructor(protected value: string) {}

  public toString() {
    return String(this.value);
  }

  public is(value: Enum | string) {
    return this.value = value.toString();
  }
}

class SourceEnum extends Enum {
  public static value1 = new SourceEnum('value1');
  public static value2 = new SourceEnum('value2');
}

class TestEnum extends SourceEnum {
  public static value3 = new TestEnum('value3');
  public static value4 = new TestEnum('value4');
```

```
}

function check(test: TestEnum) {
  return test === TestEnum.value2;
}

let value1 = TestEnum.value1;

console.log(value1 + 'hello');
console.log(value1.toString() === 'value1');
console.log(value1.is('value1'));
console.log(!TestEnum.value3.is(TestEnum.value3));
console.log(check(TestEnum.value2));
// this works but perhaps your TSLint would complain
// attention! does not work with ===
// use .is() instead
console.log(TestEnum.value1 == <any>'value1');
```

## Erweitern von Enums ohne benutzerdefinierte Enum-Implementierung

```
enum SourceEnum {
  value1 = <any>'value1',
  value2 = <any>'value2'
}

enum AdditionToSourceEnum {
  value3 = <any>'value3',
  value4 = <any>'value4'
}

// we need this type for TypeScript to resolve the types correctly
type TestEnumType = SourceEnum | AdditionToSourceEnum;
// and we need this value "instance" to use values
let TestEnum = Object.assign({}, SourceEnum, AdditionToSourceEnum);
// also works fine the TypeScript 2 feature
// let TestEnum = { ...SourceEnum, ...AdditionToSourceEnum };

function check(test: TestEnumType) {
  return test === TestEnum.value2;
}

console.log(TestEnum.value1);
console.log(TestEnum.value2 === <any>'value2');
console.log(check(TestEnum.value2));
console.log(check(TestEnum.value3));
```

Aufzählungen online lesen: https://riptutorial.com/de/typescript/topic/4954/aufzahlungen

# Kapitel 4: Benutzerdefinierte Typenschutz

## Syntax

- typeof x === "Typname"
- x Instanz von TypeName
- Funktion (foo: any): foo ist TypeName {/ * code und liefert boolesche Werte * /}

## Bemerkungen

Bei der Verwendung von Typanmerkungen in TypeScript-Einschränkungen müssen die möglichen Typen behandelt werden, mit denen Ihr Code umgehen muss. Es ist jedoch weiterhin üblich, verschiedene Codepfade zu verwenden, die auf dem Laufzeittyp einer Variablen basieren.

Mit Typwächtern können Sie Code schreiben, der auf Grundlage des Laufzeittyps einer Variablen unterscheidet. Dabei bleiben Sie stark typisiert und vermeiden Umwandlungen (auch als Typassertionen bezeichnet).

## Examples

### Instanceof verwenden

`instanceof` setzt `instanceof` , dass die Variable den Typ `any` .

Dieser Code ( probiere es ):

```
class Pet { }
class Dog extends Pet {
    bark() {
        console.log("woof");
    }
}
class Cat extends Pet {
    purr() {
        console.log("meow");
    }
}

function example(foo: any) {
    if (foo instanceof Dog) {
        // foo is type Dog in this block
        foo.bark();
    }

    if (foo instanceof Cat) {
        // foo is type Cat in this block
        foo.purr();
    }
}
```

```
example(new Dog());
example(new Cat());
```

druckt

```
woof
meom
```

zur Konsole.

### Mit typeof

`typeof` wird verwendet, wenn Sie zwischen den Typen `number` , `string` , `boolean` und `symbol` . Andere String-Konstanten werden nicht fehlerhaft sein, werden aber auch nicht zum Einschränken von Typen verwendet.

Im Gegensatz zu `instanceof` arbeitet `typeof` mit einer beliebigen Variable. Im folgenden Beispiel könnte `foo` als `number | string` eingegeben werden `number | string` ohne Ausgabe.

Dieser Code ( probiere es ):

```
function example(foo: any) {
    if (typeof foo === "number") {
        // foo is type number in this block
        console.log(foo + 100);
    }

    if (typeof foo === "string") {
        // fooi is type string in this block
        console.log("not a number: " + foo);
    }
}

example(23);
example("foo");
```

druckt

```
123
not a number: foo
```

### Typüberwachungsfunktionen

Sie können Funktionen, die als Typenwächter dienen, mit einer beliebigen Logik deklarieren.

Sie nehmen die Form an:

```
function functionName(variableName: any): variableName is DesiredType {
    // body that returns boolean
}
```

Wenn die Funktion true zurückgibt, wird der Typ von `DesiredType` in jedem Block, der von einem Aufruf der Funktion überwacht wird, auf `DesiredType` .

Zum Beispiel ( probiere es ):

```
function isString(test: any): test is string {
    return typeof test === "string";
}

function example(foo: any) {
    if (isString(foo)) {
        // foo is type as a string in this block
        console.log("it's a string: " + foo);
    } else {
        // foo is type any in this block
        console.log("don't know what this is! [" + foo + "]");
    }
}

example("hello world");          // prints "it's a string: hello world"
example({ something: "else" });  // prints "don't know what this is! [[object Object]]"
```

Ein Prädikat für einen Funktionstyp eines Guard (das `foo is Bar` in der Funktion return type position) wird zur Kompilierzeit verwendet, um Typen einzuengen. Das Typ-Prädikat und die Funktion müssen übereinstimmen, sonst funktioniert Ihr Code nicht.

Type-Guard-Funktionen müssen nicht `typeof` oder `instanceof` , sie können eine kompliziertere Logik verwenden.

Dieser Code bestimmt beispielsweise, ob Sie ein jQuery-Objekt haben, indem Sie nach der Versionszeichenfolge suchen.

```
function isJQuery(foo): foo is JQuery {
    // test for jQuery's version string
    return foo.jquery !== undefined;
}

function example(foo) {
    if (isJQuery(foo)) {
        // foo is typed JQuery here
        foo.eq(0);
    }
}
```

Benutzerdefinierte Typenschutz online lesen:
https://riptutorial.com/de/typescript/topic/8034/benutzerdefinierte-typenschutz

# Kapitel 5: Debuggen

## Einführung

Es gibt zwei Möglichkeiten, TypeScript auszuführen und zu debuggen:

**Transpile to JavaScript** , führen Sie im Knoten aus und verwenden Sie Zuordnungen, um eine Verknüpfung zu den TypeScript-Quelldateien herzustellen

oder

**Führen Sie TypeScript direkt** mit ts-node aus

In diesem Artikel werden beide Möglichkeiten beschrieben, die Visual Studio Code und WebStorm verwenden . Alle Beispiele setzen *voraus* , dass Ihre Hauptdatei *index.ts ist* .

## Examples

### JavaScript mit SourceMaps in Visual Studio-Code

In der `tsconfig.json` eingestellt

```
"sourceMap": true,
```

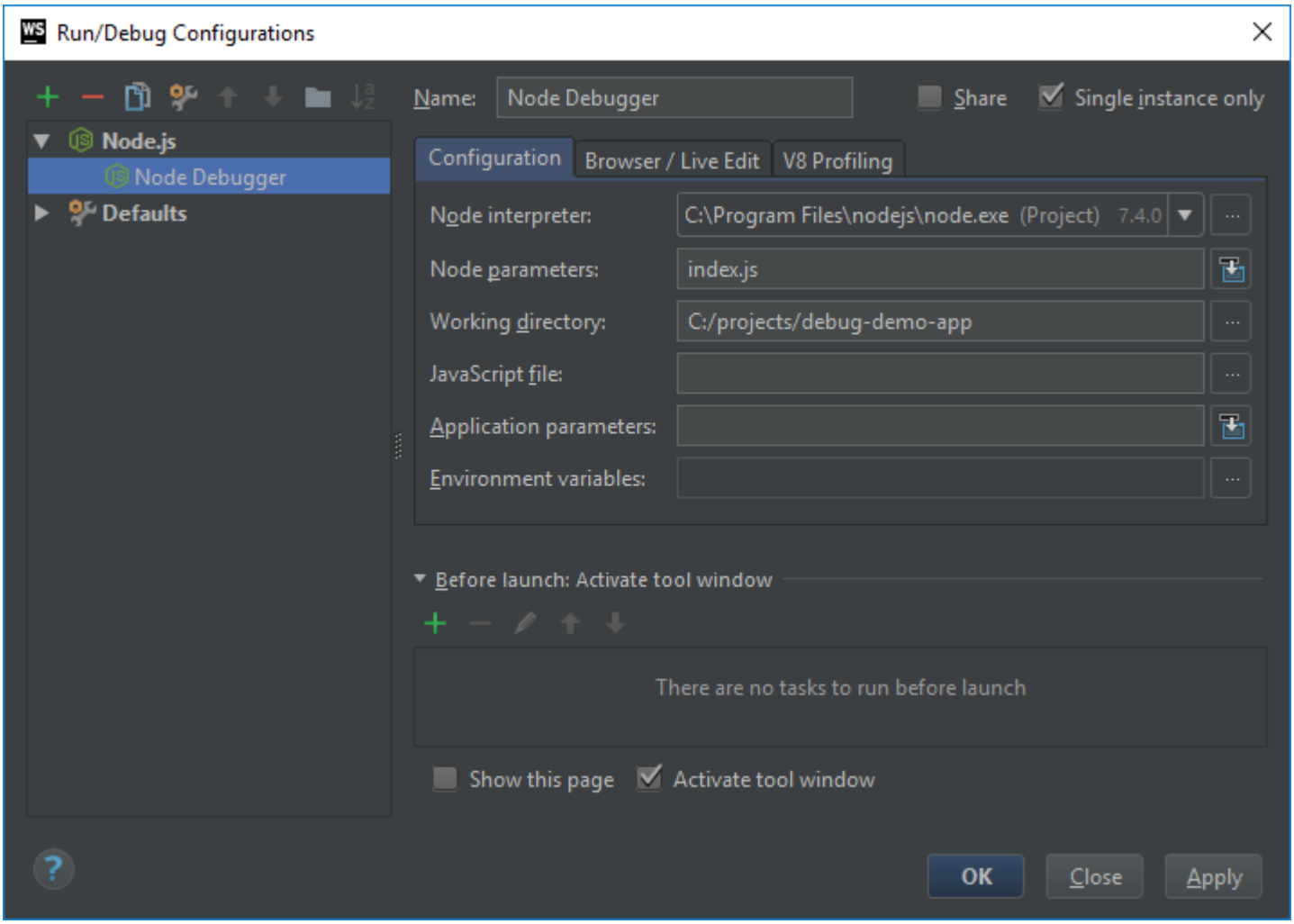
um mit den js-Dateien Zuordnungen aus den TypeScript-Quellen mit dem Befehl `tsc` zu generieren.
Die Datei launch.json :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
            "request": "launch",
            "name": "Launch Program",
            "program": "${workspaceRoot}\\index.js",
            "cwd": "${workspaceRoot}",
            "outFiles": [],
            "sourceMaps": true
        }
    ]
}
```

Dies startet den Knoten mit der generierten Datei index.js (wenn Ihre Hauptdatei index.ts ist) und dem Debugger in Visual Studio Code, der Haltepunkte hält und Variablenwerte innerhalb Ihres TypeScript-Codes auflöst.

### JavaScript mit SourceMaps in WebStorm

Erstellen Sie eine *Node.js* Debug - `index.js` Konfiguration und verwenden `index.js` als *Knoten Parameter.*




## TypeScript mit TS-Knoten in Visual Studio-Code

Fügen Sie Ihrem TypeScript-Projekt ts-node hinzu:

```
npm i ts-node
```

Fügen Sie Ihrem `package.json` ein Skript `package.json` :

```
"start:debug": "ts-node --inspect=5858 --debug-brk --ignore false index.ts"
```

Die `launch.json` muss so konfiguriert sein, dass sie den *node2*- Typ verwendet und npm startet, um das `start:debug` Skript *auszuführen* :

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node2",
            "request": "launch",
            "name": "Launch Program",
```

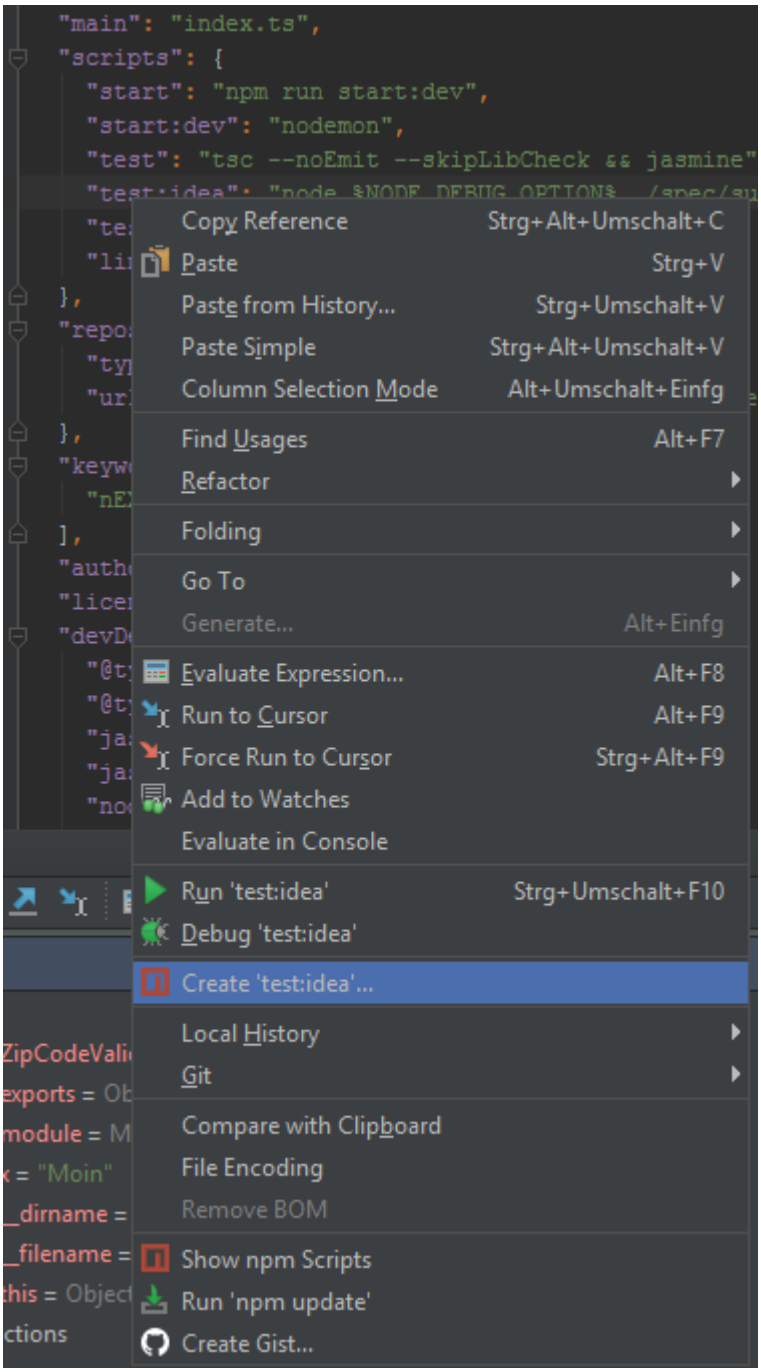
```
            "runtimeExecutable": "npm",
            "windows": {
                "runtimeExecutable": "npm.cmd"
            },
            "runtimeArgs": [
                "run-script",
                "start:debug"
            ],
            "cwd": "${workspaceRoot}/server",
            "outFiles": [],
            "port": 5858,
            "sourceMaps": true
        }
    ]
}
```

## TypeScript mit TS-Knoten in WebStorm

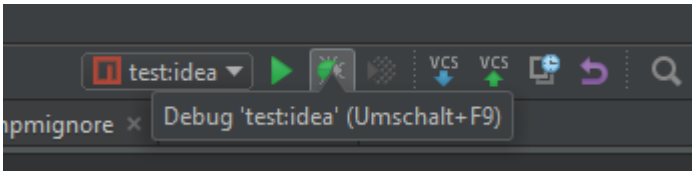
Fügen Sie dieses `package.json` zu Ihrer `package.json` :

```
"start:idea": "ts-node %NODE_DEBUG_OPTION% --ignore false index.ts",
```

Klicken Sie mit der rechten Maustaste auf das Skript und wählen Sie *Create 'test: idea' ...* und bestätigen Sie mit 'OK', um die Debug-Konfiguration zu erstellen:




Starten Sie den Debugger mit dieser Konfiguration:




Debuggen online lesen: https://riptutorial.com/de/typescript/topic/9131/debuggen

# Kapitel 6: Externe Bibliotheken importieren

## Syntax

- `import {component} from 'libName'; // Will import the class "component"`
- `import {component as c} from 'libName'; // Will import the class "component" into a "c" object`
- `import component from 'libname'; // Will import the default export from libName`
- `import * as lib from 'libName'; // Will import everything from libName into a "lib" object`
- `import lib = require('libName'); // Will import everything from libName into a "lib" object`
- `const lib: any = require('libName'); // Will import everything from libName into a "lib" object`
- `import 'libName'; // Will import libName module for its side effects only`

## Bemerkungen

Es könnte scheinen, dass die Syntax

```
import * as lib from 'libName';
```

und

```
import lib = require('libName');
```

sind dasselbe, aber nicht!

Betrachten wir , dass wir eine Klasse **Person** exportiert mit Typoskript spezifischen importieren möchten `export =` Syntax:

```
class Person {
...
}
export = Person;
```

In diesem Fall ist ein Import mit es6-Syntax nicht möglich (wir würden zur Kompilierzeit einen Fehler erhalten). TypScript-spezifische `import =` -Syntax muss verwendet werden.

```
import * as Person from 'Person';  //compile error
import Person = require('Person');  //OK
```

Das Gegenteil ist der Fall: Klassische Module können mit der zweiten Syntax importiert werden. Die letzte Syntax ist gewissermaßen leistungsfähiger, da sie alle Exporte importieren kann.

Weitere Informationen finden Sie in der offiziellen Dokumentation .

## Examples

## Ein Modul von npm importieren

Wenn Sie eine Typdefinitionsdatei (d.ts) für das Modul haben, können Sie eine `import` .

```
import _ = require('lodash');
```

Wenn Sie keine Definitionsdatei für das Modul haben, gibt TypeScript bei der Kompilierung einen Fehler aus, da das zu importierende Modul nicht gefunden wird.

In diesem Fall können Sie das Modul mit der normalen Laufzeit importieren `require` Funktion. Dies gibt es jedoch als `any` Typ zurück.

```
// The _ variable is of type any, so TypeScript will not perform any type checking.
const _: any = require('lodash');
```

Ab TypeScript 2.0 können Sie auch eine *Umgebungsmoduldeklaration* verwenden, um TypeScript mitzuteilen, dass ein Modul vorhanden ist, wenn Sie keine Typdefinitionsdatei für das Modul haben. In diesem Fall kann TypeScript jedoch keine aussagekräftige Typüberprüfung liefern.

```
declare module "lodash";

// you can now import from lodash in any way you wish:
import { flatten } from "lodash";
import * as _ from "lodash";
```

Seit TypeScript 2.1 wurden die Regeln noch weiter gelockert. Solange ein Modul in Ihrem Verzeichnis `node_modules` vorhanden ist, können Sie es `node_modules` mit TypeScript importieren, auch wenn keine Moduldeklaration vorliegt. (Beachten Sie, dass bei Verwendung der Compileroption `--noImplicitAny` die folgende `--noImplicitAny` immer noch eine Warnung generiert.)

```
// Will work if `node_modules/someModule/index.js` exists, or if
`node_modules/someModule/package.json` has a valid "main" entry point
import { foo } from "someModule";
```

## Definitionsdateien suchen

für Typoskript 2.x:

Definitionen von DefinitelyTyped sind über das @types npm- Paket verfügbar

```
npm i --save lodash
npm i --save-dev @types/lodash
```

Falls Sie jedoch Typen von anderen Repos verwenden möchten, können Sie die alte Methode verwenden:

für Typoskript 1.x:

Typings ist ein npm-Paket, mit dem Typendefinitionsdateien automatisch in einem lokalen Projekt

installiert werden können. Ich empfehle Ihnen, den Schnellstart zu lesen.

```
npm install -global typings
```

Nun haben wir Zugriff auf die Typisierungen cli.

1. Der erste Schritt besteht darin, nach dem vom Projekt verwendeten Paket zu suchen

```
typings search lodash
NAME              SOURCE HOMEPAGE                                          DESCRIPTION
VERSIONS UPDATED
lodash            dt     http://lodash.com/                                          2
2016-07-20T00:13:09.000Z
lodash            global                                                             1
2016-07-01T20:51:07.000Z
lodash            npm    https://www.npmjs.com/package/lodash                        1
2016-07-01T20:51:07.000Z
```

2. Dann entscheiden Sie, aus welcher Quelle Sie installieren sollen. Ich verwende dt, was für DefinitelyTyped steht, ein GitHub-Repo, in dem die Community Typisierungen bearbeiten kann. Normalerweise wird auch das zuletzt aktualisiert.

3. Installieren Sie die Typisierungsdateien

```
typings install dt~lodash --global --save
```

Brechen wir den letzten Befehl auf. Wir installieren die DefinitelyTyped-Version von lodash als globale Typisierungsdatei in unserem Projekt und speichern sie als Abhängigkeit in der `typings.json` . Wo immer wir lodash importieren, lädt typscript die lodash-Typisierungsdatei.

4. Wenn Sie Typisierungen installieren möchten, die nur für die Entwicklungsumgebung verwendet werden, können Sie das `--save-dev` :

```
typings install chai --save-dev
```

## Verwendung globaler externer Bibliotheken ohne Typisierung

Obwohl Module ideal sind, können Sie, wenn auf die von Ihnen verwendete Bibliothek durch eine globale Variable (wie $ oder _) verwiesen wird, da sie durch ein `script` geladen wurde, eine Umgebungsdeklaration erstellen, um darauf zu verweisen:

```
declare const _: any;
```

## Definitionsdateien mit Typoscript 2.x suchen

Mit den 2.x-Versionen von Typescript sind jetzt Typisierungen im Repository npm @types verfügbar. Diese werden vom Typenscript-Compiler automatisch aufgelöst und sind viel einfacher zu verwenden.

Um eine Typdefinition zu installieren, installieren Sie sie einfach als dev-Abhängigkeit in Ihrer

Projekte package.json

z.B

```
npm i -S lodash
npm i -D @types/lodash
```

Nach der Installation verwenden Sie das Modul einfach wie zuvor

```
import * as _ from 'lodash'
```

Externe Bibliotheken importieren online lesen:
https://riptutorial.com/de/typescript/topic/1542/externe-bibliotheken-importieren

# Kapitel 7: Funktionen

## Bemerkungen

Link zur TypScript-Dokumentation für Funktionen

## Examples

### Optionale und Standardparameter

#### Optionale Parameter

In TypeScript wird davon ausgegangen, dass jeder Parameter von der Funktion benötigt wird. Sie können ein hinzufügen **?** am Ende eines Parameternamens, um ihn als optional festzulegen.

Der `lastName` Parameter dieser Funktion ist beispielsweise optional:

```
function buildName(firstName: string, lastName?: string) {
    // ...
}
```

Optionale Parameter müssen hinter allen nicht optionalen Parametern stehen:

```
function buildName(firstName?: string, lastName: string) // Invalid
```

#### Standardparameter

Wenn der Benutzer `undefined` übergibt oder kein Argument angibt, wird der Standardwert zugewiesen. Diese Parameter werden als *Standardinitialisierungsparameter bezeichnet* .

Beispielsweise ist "Smith" der Standardwert für den `lastName` Parameter.

```
function buildName(firstName: string, lastName = "Smith") {
    // ...
}
buildName('foo', 'bar');      // firstName == 'foo', lastName == 'bar'
buildName('foo');             // firstName == 'foo', lastName == 'Smith'
buildName('foo', undefined);  // firstName == 'foo', lastName == 'Smith'
```

### Arten von Funktionen

#### Benannte Funktionen

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

**Anonyme Funktionen**

```
let multiply = function(a, b) { return a * b; };
```

**Lambda / Pfeil-Funktionen**

```
let multiply = (a, b) => { return a * b; };
```

## Funktion als Parameter

Angenommen, wir möchten eine Funktion als Parameter erhalten, können wir dies folgendermaßen tun:

```
function foo(otherFunc: Function): void {
    ...
}
```

Wenn wir einen Konstruktor als Parameter erhalten wollen:

```
function foo(constructorFunc: { new() }) {
    new constructorFunc();
}

function foo(constructorWithParamsFunc: { new(num: number) }) {
    new constructorWithParamsFunc(1);
}
```

Oder um das Lesen zu erleichtern, können wir eine Schnittstelle definieren, die den Konstruktor beschreibt:

```
interface IConstructor {
    new();
}

function foo(contructorFunc: IConstructor) {
    new constructorFunc();
}
```

Oder mit Parametern:

```
interface INumberConstructor {
    new(num: number);
}

function foo(contructorFunc: INumberConstructor) {
    new contructorFunc(1);
}
```

Auch bei Generika:

```
interface ITConstructor<T, U> {
```

```
    new(item: T): U;
}

function foo<T, U>(contructorFunc: ITConstructor<T, U>, item: T): U {
    return new contructorFunc(item);
}
```

Wenn wir eine einfache Funktion und keinen Konstruktor erhalten wollen, ist das fast dasselbe:

```
function foo(func: { (): void }) {
    func();
}

function foo(constructorWithParamsFunc: { (num: number): void }) {
    new constructorWithParamsFunc(1);
}
```

Oder um das Lesen zu erleichtern, können wir eine Schnittstelle definieren, die die Funktion beschreibt:

```
interface IFunction {
    (): void;
}

function foo(func: IFunction ) {
    func();
}
```

Oder mit Parametern:

```
interface INumberFunction {
    (num: number): string;
}

function foo(func: INumberFunction ) {
    func(1);
}
```

Auch bei Generika:

```
interface ITFunc<T, U> {
    (item: T): U;
}

function foo<T, U>(contructorFunc: ITFunc<T, U>, item: T): U {
    return func(item);
}
```

## Funktionen mit Unionstypen

Eine TypeScript-Funktion kann Parameter mehrerer vordefinierter Typen aufnehmen, die Unionstypen verwenden.

```
function whatTime(hour:number|string, minute:number|string):string{
    return hour+':'+minute;
}

whatTime(1,30)          //'1:30'
whatTime('1',30)        //'1:30'
whatTime(1,'30')        //'1:30'
whatTime('1','30')      //'1:30'
```

Typescript behandelt diese Parameter als einen einzigen Typ, der eine Vereinigung der anderen Typen darstellt. Ihre Funktion muss daher in der Lage sein, Parameter eines beliebigen Typs zu behandeln, der in der Union enthalten ist.

```
function addTen(start:number|string):number{
    if(typeof number === 'string'){
        return parseInt(number)+10;
    }else{
        else return number+10;
    }
}
```

Funktionen online lesen: https://riptutorial.com/de/typescript/topic/1841/funktionen

# Kapitel 8: Generics

## Syntax

- Die generischen Typen, die innerhalb der eckigen Klammern deklariert sind: `<T>`
- Das Einschränken der generischen Typen erfolgt mit dem Schlüsselwort extend: `<T extends Car>`

## Bemerkungen

Die generischen Parameter stehen zur Laufzeit nicht zur Verfügung, sie dienen nur der Kompilierzeit. Das bedeutet, dass Sie so etwas nicht machen können:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (T instanceof Executable1) {    // Compilation error
            ...
        } else if (U instanceof Executable2){    // Compilation error
            ...
        }
    }
}
```

Die Klasseninformationen bleiben jedoch erhalten, sodass Sie immer noch den Typ einer Variablen testen können, wie Sie es immer tun konnten:

```
class Executor<T, U> {
    public execute(executable: T): void {
        if (executable instanceof Executable1) {
            ...
        } else if (executable instanceof Executable2){
            ...
        } // But in this method, since there is no parameter of type `U` it is non-sensical to
ask about U's "type"
    }
}
```

## Examples

### Generische Schnittstellen

# Eine generische Schnittstelle deklarieren

```
interface IResult<T> {
    wasSuccessfull: boolean;
    error: T;
}
```

```
var result: IResult<string> = ....
var error: string = result.error;
```

# Generische Schnittstelle mit mehreren Typparametern

```
interface IRunnable<T, U> {
    run(input: T): U;
}

var runnable: IRunnable<string, number> = ...
var input: string;
var result: number = runnable.run(input);
```

# Implementieren einer generischen Schnittstelle

```
interface IResult<T>{
    wasSuccessfull: boolean;
    error: T;

    clone(): IResult<T>;
}
```

Implementiere es mit generischer Klasse:

```
class Result<T> implements IResult<T> {
    constructor(public result: boolean, public error: T) {
    }

    public clone(): IResult<T> {
        return new Result<T>(this.result, this.error);
    }
}
```

Implementiere es mit einer nicht generischen Klasse:

```
class StringResult implements IResult<string> {
    constructor(public result: boolean, public error: string) {
    }

    public clone(): IResult<string> {
        return new StringResult(this.result, this.error);
    }
}
```

## Generische Klasse

```
class Result<T> {
    constructor(public wasSuccessful: boolean, public error: T) {
    }

    public clone(): Result<T> {
       ...
    }
}

let r1 = new Result(false, 'error: 42');  // Compiler infers T to string
let r2 = new Result(false, 42);           // Compiler infers T to number
let r3 = new Result<string>(true, null);  // Explicitly set T to string
let r4 = new Result<string>(true, 4);     // Compilation error because 4 is not a string
```

## Generics-Einschränkungen

Einfache Einschränkung:

```
interface IRunnable {
    run(): void;
}

interface IRunner<T extends IRunnable> {
    runSafe(runnable: T): void;
}
```

Komplexere Einschränkung:

```
interface IRunnble<U> {
    run(): U;
}

interface IRunner<T extends IRunnable<U>, U> {
    runSafe(runnable: T): U;
}
```

Noch komplexer:

```
interface IRunnble<V> {
    run(parameter: U): V;
}

interface IRunner<T extends IRunnable<U, V>, U, V> {
    runSafe(runnable: T, parameter: U): V;
}
```

Inline-Typeinschränkungen:

```
interface IRunnable<T extends { run(): void }> {
    runSafe(runnable: T): void;
}
```

## Generische Funktionen

In Schnittstellen:

```
interface IRunner {
    runSafe<T extends IRunnable>(runnable: T): void;
}
```

In Klassen:

```
class Runner implements IRunner {

    public runSafe<T extends IRunnable>(runnable: T): void {
        try {
            runnable.run();
        } catch(e) {
        }
    }

}
```

Einfache Funktionen:

```
function runSafe<T extends IRunnable>(runnable: T): void {
    try {
        runnable.run();
    } catch(e) {
    }
}
```

## Verwenden generischer Klassen und Funktionen:

Generische Klasseninstanz erstellen:

```
var stringRunnable = new Runnable<string>();
```

Generische Funktion ausführen:

```
function runSafe<T extends Runnable<U>, U>(runnable: T);

// Specify the generic types:
runSafe<Runnable<string>, string>(stringRunnable);

// Let typescript figure the generic types by himself:
runSafe(stringRunnable);
```

## Geben Sie Parameter als Einschränkungen ein

Mit TypeScript 1.8 wird es möglich, dass eine Typparametereinschränkung auf Typparameter aus derselben Typparameterliste verweist. Bisher war dies ein Fehler.

```
 function assign<T extends U, U>(target: T, source: U): T {
    for (let id in source) {
        target[id] = source[id];
    }
    return target;
}

let x = { a: 1, b: 2, c: 3, d: 4 };
assign(x, { b: 10, d: 20 });
assign(x, { e: 0 });  // Error
```

Generics online lesen: https://riptutorial.com/de/typescript/topic/2132/generics

# Kapitel 9: Integration mit Build-Tools

## Bemerkungen

Weitere Informationen finden Sie in den offiziellen Webseiten- Typoskripten, die in Build-Tools integriert werden

## Examples

### Installieren und konfigurieren Sie Webpack + -Ladegeräte

Installation

```
npm install -D webpack typescript ts-loader
```

webpack.config.js

```
module.exports = {
  entry: {
    app: ['./src/'],
  },
  output: {
    path: __dirname,
    filename: './dist/[name].js',
  },
  resolve: {
    extensions: ['', '.js', '.ts'],
  },
  module: {
    loaders: [{
      test: /\.ts(x)$/, loaders: ['ts-loader'], exclude: /node_modules/
    }],
  }
};
```

### Browserify

# Installieren

```
npm install tsify
```

# Befehlszeilenschnittstelle verwenden

```
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

# API verwenden

```
var browserify = require("browserify");
var tsify = require("tsify");

browserify()
  .add("main.ts")
  .plugin("tsify", { noImplicitAny: true })
  .bundle()
  .pipe(process.stdout);
```

Weitere Details: smrq / tsify

### Grunzen

# Installieren

```
npm install grunt-ts
```

# Grundlegende Gruntfile.js

```
module.exports = function(grunt) {
    grunt.initConfig({
        ts: {
            default : {
                src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
    grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

Weitere Details: TypeStrong / grunt-ts

### Schluck

# Installieren

```
npm install gulp-typescript
```

# Grundlegende gulpfile.js

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
    var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
              noImplicitAny: true,
              out: "output.js"
        }));
    return tsResult.js.pipe(gulp.dest("built/local"));
});
```

# gulpfile.js mit einer vorhandenen tsconfig.json

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

var tsProject = ts.createProject('tsconfig.json', {
    noImplicitAny: true // You can add and overwrite parameters here
});

gulp.task("default", function () {
    var tsResult = tsProject.src()
        .pipe(tsProject());
    return tsResult.js.pipe(gulp.dest('release'));
});
```

Weitere Details: ivogabe / gulp-typescript

### Webpack

# Installieren

```
npm install ts-loader --save-dev
```

# Grundlegende webpack.config.js

## Webpack 2.x, 3.x

```
module.exports = {
    resolve: {
        extensions: ['.ts', '.tsx', '.js']
    },
    module: {
        rules: [
            {
```

```
                // Set up ts-loader for .ts/.tsx files and exclude any imports from
node_modules.
                test: /\.tsx?$/,
                loaders: ['ts-loader'],
                exclude: /node_modules/
            }
        ]
    },
    entry: [
        // Set index.tsx as application entry point.
        './index.tsx'
    ],
    output: {
      filename: "bundle.js"
    }
};
```

# Webpack 1.x

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js"
    },
    resolve: {
        // Add '.ts' and '.tsx' as a resolvable extension.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },
    module: {
        loaders: [
            // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
            { test: /\.ts(x)?$/, loader: "ts-loader", exclude: /node_modules/ }
        ]
    }
}
```

Weitere Details zum TS-Loader finden Sie hier .

Alternativen:

- Awesome-Typoskript-Ladeprogramm

## MSBuild

Aktualisieren Sie die Projektdatei so, dass sie lokal installierte `Microsoft.TypeScript.Default.props` (oben) und `Microsoft.TypeScript.targets` (unten) enthält:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Include default props at the bottom -->
  <Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft
```

```
Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>

  <!-- TypeScript configurations go here -->
  <PropertyGroup Condition="'$(Configuration)' == 'Debug'">
    <TypeScriptRemoveComments>false</TypeScriptRemoveComments>
    <TypeScriptSourceMap>true</TypeScriptSourceMap>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)' == 'Release'">
    <TypeScriptRemoveComments>true</TypeScriptRemoveComments>
    <TypeScriptSourceMap>false</TypeScriptSourceMap>
  </PropertyGroup>

  <!-- Include default targets at the bottom -->
  <Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microsoft


Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript
/>
</Project>
```

Weitere Informationen zum Definieren von MSBuild-Compileroptionen: Festlegen von Compileroptionen in MSBuild-Projekten

## NuGet

- Rechtsklick -> NuGet-Pakete verwalten
- Suchen Sie nach `Microsoft.TypeScript.MSBuild`
- Klicken Sie auf `Install`
- Wenn die Installation abgeschlossen ist, bauen Sie neu auf!

Weitere Informationen finden Sie im Package Manager-Dialog und bei Verwendung nächtlicher Builds mit NuGet

Integration mit Build-Tools online lesen: https://riptutorial.com/de/typescript/topic/2860/integration-mit-build-tools

# Kapitel 10: Klasse Dekorateur

## Parameter

| Parameter | Einzelheiten |
|---|---|
| Ziel | Die Klasse wird dekoriert |

## Examples

### Grundklasse Dekorateur

Ein Klassendekorateur ist nur eine Funktion, die die Klasse als einziges Argument aufnimmt und zurückgibt, nachdem sie etwas getan hat:

```
function log<T>(target: T) {

    // Do something with target
    console.log(target);

    // Return target
    return target;

}
```

Dann können wir den Klassendekorateur auf eine Klasse anwenden:

```
@log
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

### Generierung von Metadaten mit einem Klassendekorateur

Dieses Mal werden wir einen Klassendekorateur deklarieren, der einer Klasse einige Metadaten hinzufügt, wenn wir ihn anwenden:

```
function addMetadata(target: any) {

    // Add some metadata
    target.__customMetadata = {
        someKey: "someValue"
    };
```

```
    // Return target
    return target;

}
```

Wir können dann den Klassendekorateur anwenden:

```
@addMetadata
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}

function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromClass(Person));
```

Der Dekorator wird angewendet, wenn die Klasse nicht deklariert wird, nicht wenn wir Instanzen der Klasse erstellen. Dies bedeutet, dass die Metadaten von allen Instanzen einer Klasse gemeinsam genutzt werden:

```
function getMetadataFromInstance(target: any) {
    return target.constructor.__customMetadata;
}

let person1 = new Person("John");
let person2 = new Person("Lisa");

console.log(getMetadataFromInstance(person1));
console.log(getMetadataFromInstance(person2));
```

## Argumente an einen Klassendekorateur übergeben

Wir können einen Klassendekorateur mit einer anderen Funktion umschließen, um die Anpassung zu ermöglichen:

```
function addMetadata(metadata: any) {
    return function log(target: any) {

        // Add metadata
        target.__customMetadata = metadata;

        // Return target
        return target;

    }
}
```

Das `addMetadata` nimmt einige als Konfiguration verwendete Argumente und gibt dann eine unbenannte Funktion zurück, die der eigentliche Dekorateur ist. Im Dekorateur können wir auf die Argumente zugreifen, da eine Schließung vorhanden ist.

Wir können den Dekorator dann aufrufen, indem er einige Konfigurationswerte übergibt:

```
@addMetadata({ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" })
class Person {
    private _name: string;
    public constructor(name: string) {
        this._name = name;
    }
    public greet() {
        return this._name;
    }
}
```

Wir können die folgende Funktion verwenden, um auf die generierten Metadaten zuzugreifen:

```
function getMetadataFromClass(target: any) {
    return target.__customMetadata;
}

console.log(getMetadataFromInstance(Person));
```

Wenn alles richtig lief, sollte die Konsole Folgendes anzeigen:

```
{ guid: "417c6ec7-ec05-4954-a3c6-73a0d7f9f5bf" }
```

Klasse Dekorateur online lesen: https://riptutorial.com/de/typescript/topic/4592/klasse-dekorateur

# Kapitel 11: Klassen

## Einführung

TypeScript unterstützt wie ECMA Script 6 die objektorientierte Programmierung mit Klassen. Dies steht im Gegensatz zu älteren JavaScript-Versionen, die nur prototypbasierte Vererbungskette unterstützen.

Die Klassenunterstützung in TypeScript ähnelt der von Sprachen wie Java und C #, da Klassen von anderen Klassen erben können, während Objekte als Klasseninstanzen instanziiert werden.

Ähnlich wie diese Sprachen können TypeScript-Klassen Schnittstellen implementieren oder Generics verwenden.

## Examples

### Einfache Klasse

```
class Car {
    public position: number = 0;
    private speed: number = 42;

    move() {
        this.position += this.speed;
    }
}
```

In diesem Beispiel deklarieren wir eine einfache Klasse `Car` . Die Klasse besteht aus drei Mitgliedern: ein Privateigentum `speed` , eine öffentliche Eigenschaft `position` und eine öffentliche Methode `move` . Beachten Sie, dass jedes Mitglied standardmäßig öffentlich ist. Deshalb ist `move()` öffentlich, auch wenn wir das Schlüsselwort `public` nicht verwendet haben.

```
var car = new Car();        // create an instance of Car
car.move();                 // call a method
console.log(car.position);  // access a public property
```

### Grundvererbung

```
class Car {
    public position: number = 0;
    protected speed: number = 42;

    move() {
        this.position += this.speed;
    }
}

class SelfDrivingCar extends Car {
```

```
    move() {
        // start moving around :-)
        super.move();
        super.move();
    }
}
```

Dieses Beispiel zeigt , wie eine sehr einfache Unterklasse der erstellen `Car` - Klasse des `extends` Schlüsselwort. Die `SelfDrivingCar` Klasse überschreibt die `move()` Methode und verwendet die Implementierung der Basisklasse mit `super` .

## Konstrukteure

In diesem Beispiel verwenden wir den `constructor` eine öffentliche Eigenschaft zu erklären `position` und eine geschützte Eigenschaft `speed` in der Basisklasse. Diese Eigenschaften werden als *Parametereigenschaften bezeichnet* . Sie lassen uns einen Konstruktorparameter und ein Member an einer Stelle deklarieren.

Eines der besten Dinge in TypeScript ist die automatische Zuweisung von Konstruktorparametern an die entsprechende Eigenschaft.

```
class Car {
    public position: number;
    protected speed: number;

    constructor(position: number, speed: number) {
        this.position = position;
        this.speed = speed;
    }

    move() {
        this.position += this.speed;
    }
}
```

Der gesamte Code kann in einem einzigen Konstruktor fortgesetzt werden:

```
class Car {
    constructor(public position: number, protected speed: number) {}

    move() {
        this.position += this.speed;
    }
}
```

Beide werden mit gleichem Ergebnis von TypeScript (Entwurfszeit und Kompilierzeit) nach JavaScript übertragen, wobei jedoch deutlich weniger Code geschrieben wird:

```
var Car = (function () {
    function Car(position, speed) {
        this.position = position;
        this.speed = speed;
    }
```

```
    Car.prototype.move = function () {
        this.position += this.speed;
    };
    return Car;
}());
```

Konstruktoren abgeleiteter Klassen müssen den Basisklassenkonstruktor mit `super()` aufrufen.

```
class SelfDrivingCar extends Car {
    constructor(startAutoPilot: boolean) {
        super(0, 42);
        if (startAutoPilot) {
            this.move();
        }
    }
}

let car = new SelfDrivingCar(true);
console.log(car.position);  // access the public property position
```

## Accessoren

In diesem Beispiel ändern wir das Beispiel "Einfache Klasse", um den Zugriff auf die `speed` zu ermöglichen. Mit typecript-Accessoren können wir zusätzlichen Code in Gettern oder Setters hinzufügen.

```
class Car {
    public position: number = 0;
    private _speed: number = 42;
    private _MAX_SPEED = 100

    move() {
        this.position += this._speed;
    }

    get speed(): number {
        return this._speed;
    }

    set speed(value: number) {
        this._speed = Math.min(value, this._MAX_SPEED);
    }
}

let car = new Car();
car.speed = 120;
console.log(car.speed);  // 100
```

## Abstrakte Klassen

```
abstract class Machine {
    constructor(public manufacturer: string) {
    }

    // An abstract class can define methods of it's own, or...
```

```
    summary(): string {
        return `${this.manufacturer} makes this machine.`;
    }

    // Require inheriting classes to implement methods
    abstract moreInfo(): string;
}

class Car extends Machine {
    constructor(manufacturer: string, public position: number, protected speed: number) {
        super(manufacturer);
    }

    move() {
        this.position += this.speed;
    }

    moreInfo() {
        return `This is a car located at ${this.position} and going ${this.speed}mph!`;
    }
}

let myCar = new Car("Konda", 10, 70);
myCar.move(); // position is now 80
console.log(myCar.summary()); // prints "Konda makes this machine."
console.log(myCar.moreInfo()); // prints "This is a car located at 80 and going 70mph!"
```

Abstrakte Klassen sind Basisklassen, von denen andere Klassen ausgehen können. Sie können nicht selbst instanziiert werden (dh Sie **können keine** `new Machine("Konda")` erstellen).

Die zwei Hauptmerkmale einer abstrakten Klasse in Typescript sind:

1. Sie können eigene Methoden implementieren.
2. Sie können Methoden definieren, die erben Klassen implementieren **müssen** .

Aus diesem Grund können abstrakte Klassen konzeptionell als **Kombination aus einer Schnittstelle und einer Klasse betrachtet werden** .

## Affe patchen eine Funktion in eine vorhandene Klasse

Manchmal ist es nützlich, eine Klasse um neue Funktionen erweitern zu können. Nehmen wir zum Beispiel an, dass ein String in einen Camel-Case-String konvertiert werden soll. Daher müssen wir TypeScript mitteilen, dass `String` eine Funktion namens `toCamelCase` , die einen `string` zurückgibt.

```
interface String {
    toCamelCase(): string;
}
```

Jetzt können wir diese Funktion in die `String` Implementierung implementieren.

```
String.prototype.toCamelCase = function() : string {
    return this.replace(/[^a-z ]/ig, '')
        .replace(/(?:^\w|[A-Z]|\b\w|\s+)/g, (match: any, index: number) => {
            return +match === 0 ? "" : match[index === 0 ? 'toLowerCase' : 'toUpperCase']();
        });
```

```
}
```

Wenn diese Erweiterung von `String` geladen ist, kann sie folgendermaßen verwendet werden:

```
"This is an example".toCamelCase();    // => "thisIsAnExample"
```

### Transpilation

Bei einer Klasse `SomeClass` wir, wie das TypeScript in JavaScript umgesetzt wird.

# TypeScript-Quelle

```
class SomeClass {

    public static SomeStaticValue: string = "hello";
    public someMemberValue: number = 15;
    private somePrivateValue: boolean = false;

    constructor () {
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }

    public static getGoodbye(): string {
        return "goodbye!";
    }

    public getFortyTwo(): number {
        return 42;
    }

    private getTrue(): boolean {
        return true;
    }

}
```

# JavaScript-Quelle

Bei der Verwendung von TypeScript `v2.2.2` die Ausgabe folgendermaßen aus:

```
var SomeClass = (function () {
    function SomeClass() {
        this.someMemberValue = 15;
        this.somePrivateValue = false;
        SomeClass.SomeStaticValue = SomeClass.getGoodbye();
        this.someMemberValue = this.getFortyTwo();
        this.somePrivateValue = this.getTrue();
    }
    SomeClass.getGoodbye = function () {
        return "goodbye!";
```

```
    };
    SomeClass.prototype.getFortyTwo = function () {
        return 42;
    };
    SomeClass.prototype.getTrue = function () {
        return true;
    };
    return SomeClass;
}());
SomeClass.SomeStaticValue = "hello";
```

# Beobachtungen

- Die Modifikation des Prototyps der Klasse ist in einem IIFE eingeschlossen .
- Membervariablen sind innerhalb der Hauptklasse definiert `function` .
- Statische Eigenschaften werden direkt zum Klassenobjekt hinzugefügt, während Instanzeigenschaften zum Prototyp hinzugefügt werden.

Klassen online lesen: https://riptutorial.com/de/typescript/topic/1560/klassen

# Kapitel 12: Konfigurieren Sie das Typescript-Projekt, um alle Dateien in Typoscript zu kompilieren.

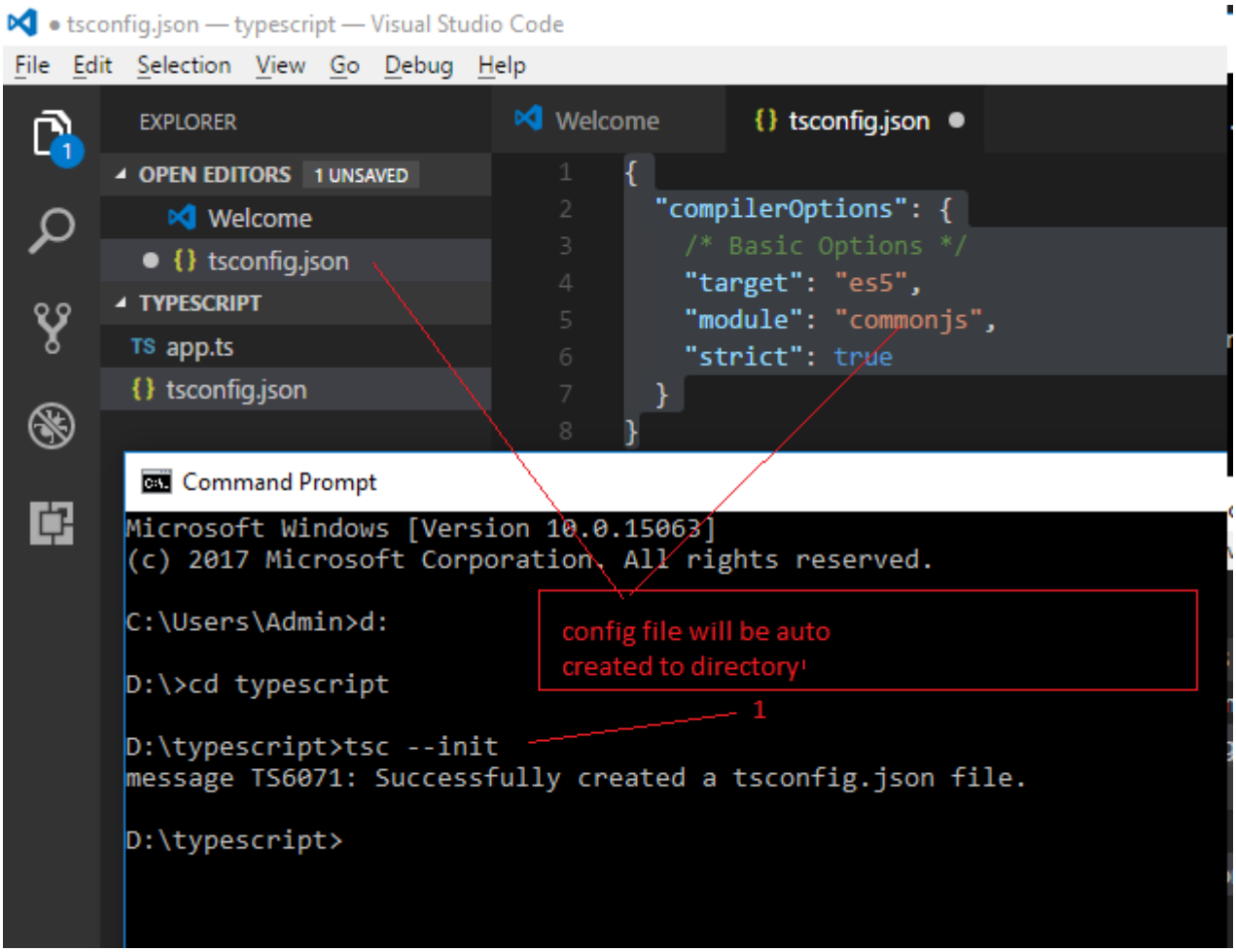
## Einführung

Erstellen Sie Ihre erste .tsconfig-Konfigurationsdatei, die dem TypeScript-Compiler die Behandlung Ihrer .ts-Dateien mitteilt

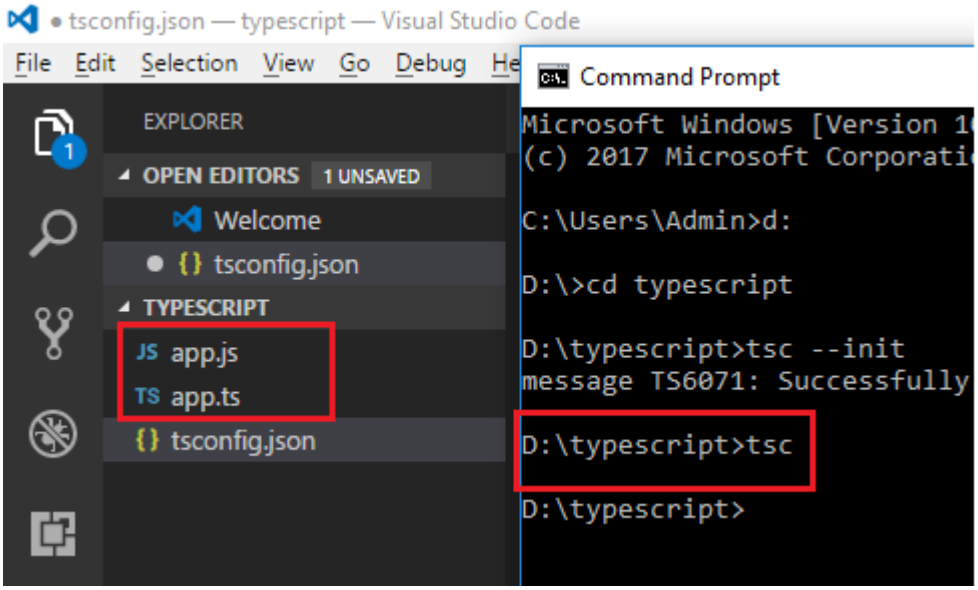
## Examples

### Setup der Typoscript-Konfigurationsdatei

- Geben Sie den Befehl " **tsc --init** " ein und **drücken Sie** die Eingabetaste.
- Vorher müssen wir die ts-Datei mit dem Befehl " **tsc app.ts** " **kompilieren.** Jetzt wird alles in der folgenden Konfigurationsdatei automatisch definiert.




- Nun können Sie alle Typoskripte mit dem Befehl " **tsc** " kompilieren. Es erstellt automatisch eine ".js" -Datei Ihrer Typoskript-Datei.




- Wenn Sie ein anderes Typoskript erstellen und den Befehl "tsc" in der Eingabeaufforderung oder im Terminal drücken, wird automatisch eine Javascript-Datei für die Typoskriptdatei erstellt.

Vielen Dank,

Konfigurieren Sie das Typescript-Projekt, um alle Dateien in Typoscript zu kompilieren. online lesen: https://riptutorial.com/de/typescript/topic/10537/konfigurieren-sie-das-typescript-projekt--um-alle-dateien-in-typoscript-zu-kompilieren-

# Kapitel 13: Mixins

## Syntax

- Klasse BeetleGuy implementiert Climbs, Bulletproof {}
- applyMixins (BeetleGuy, [Climbs, Bulletproof]);

## Parameter

| Parameter | Beschreibung |
|---|---|
| AbgeleiteterCtor | Die Klasse, die Sie als Kompositionsklasse verwenden möchten |
| baseCtors | Ein Array von Klassen, die der Kompositionsklasse hinzugefügt werden sollen |

## Bemerkungen

Bei Mixins sind drei Regeln zu beachten:

- Sie verwenden das `implements` Schlüsselwort, nicht das `extends` Schlüsselwort, wenn Sie Ihre Kompositionsklasse schreiben
- Sie benötigen eine übereinstimmende Signatur, um den Compiler leise zu halten (aber es ist keine echte Implementierung erforderlich - dies wird durch das Mixin erreicht).
- Sie müssen `applyMixins` mit den richtigen Argumenten aufrufen.

## Examples

### Beispiel für Mixins

Um Mixins zu erstellen, deklarieren Sie einfach leichte Klassen, die als "Verhalten" verwendet werden können.

```
class Flies {
    fly() {
        alert('Is it a bird? Is it a plane?');
    }
}

class Climbs {
    climb() {
        alert('My spider-sense is tingling.');
    }
}

class Bulletproof {
```

```
    deflect() {
        alert('My wings are a shield of steel.');
    }
}
```

Sie können diese Verhalten auf eine Kompositionsklasse anwenden:

```
class BeetleGuy implements Climbs, Bulletproof {
        climb: () => void;
        deflect: () => void;
}
applyMixins (BeetleGuy, [Climbs, Bulletproof]);
```

Die Funktion `applyMixins` ist für die Kompositionsarbeit erforderlich.

```
function applyMixins(derivedCtor: any, baseCtors: any[]) {
    baseCtors.forEach(baseCtor => {
        Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
             if (name !== 'constructor') {
                derivedCtor.prototype[name] = baseCtor.prototype[name];
            }
        });
    });
}
```

Mixins online lesen: https://riptutorial.com/de/typescript/topic/4727/mixins

# Kapitel 14: Module - Exportieren und Importieren

## Examples

### Hallo Weltmodul

```
//hello.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
function helloES(name: string){
    console.log(`Hola ${name}!`);
}
export {helloES};
export default hello;
```

#### Laden mit Verzeichnisindex

Wenn das Verzeichnis eine Datei mit dem Namen `index.ts` enthält, kann es nur mit dem Verzeichnisnamen geladen werden (für `index.ts` Dateiname ist optional).

```
//welcome/index.ts
export function welcome(name: string){
    console.log(`Welcome ${name}!`);
}
```

#### Beispiel für die Verwendung definierter Module

```
import {hello, helloES} from "./hello";  // load specified elements
import defaultHello from "./hello";      // load default export into name defaultHello
import * as Bundle from "./hello";       // load all exports as Bundle
import {welcome} from "./welcome";       // note index.ts is omitted

hello("World");                          // Hello World!
helloES("Mundo");                        // Hola Mundo!
defaultHello("World");                   // Hello World!

Bundle.hello("World");                   // Hello World!
Bundle.helloES("Mundo");                 // Hola Mundo!

welcome("Human");                        // Welcome Human!
```

### Deklarationen exportieren / importieren

Jede Deklaration (Variable, const, Funktion, Klasse usw.) kann vom Modul exportiert werden, um in ein anderes Modul importiert zu werden.

Typoscript bietet zwei Exporttypen: Named und Default.

### Genannter Export

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
export const unused = 0;
```

Beim Importieren benannter Exporte können Sie angeben, welche Elemente Sie importieren möchten.

```
import {hello, answerToLifeTheUniverseAndEverything} from "./adams";
hello(answerToLifeTheUniverseAndEverything);   // Hello 42!
```

### Standardexport

Jedes Modul kann einen Standardexport haben

```
// dent.ts
const defaultValue = 54;
export default defaultValue;
```

die importiert werden kann mit

```
import dentValue from "./dent";
console.log(dentValue);        // 54
```

### Gebündelter Import

Typescript bietet eine Methode zum Importieren des gesamten Moduls in eine Variable

```
// adams.ts
export function hello(name: string){
    console.log(`Hello ${name}!`);
}
export const answerToLifeTheUniverseAndEverything = 42;
```

```
import * as Bundle from "./adams";
Bundle.hello(Bundle.answerToLifeTheUniverseAndEverything);  // Hello 42!
console.log(Bundle.unused);                                 // 0
```

## Erneut exportieren

Typoskript erlaubt das erneute Exportieren von Deklarationen.

```
//Operator.ts
interface Operator {
    eval(a: number, b: number): number;
}
export default Operator;
```

```
//Add.ts
import Operator from "./Operator";
export class Add implements Operator {
    eval(a: number, b: number): number {
        return a + b;
    }
}
```

```
//Mul.ts
import Operator from "./Operator";
export class Mul implements Operator {
    eval(a: number, b: number): number {
        return a * b;
    }
}
```

Sie können alle Vorgänge in einer einzigen Bibliothek bündeln

```
//Operators.ts
import {Add} from "./Add";
import {Mul} from "./Mul";

export {Add, Mul};
```

**Benannte Deklarationen** können mit kürzerer Syntax erneut exportiert werden

```
//NamedOperators.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
```

**Standardexporte** können auch exportiert werden, es ist jedoch keine kurze Syntax verfügbar. Denken Sie daran, dass nur ein Standardexport pro Modul möglich ist.

```
//Calculator.ts
export {Add} from "./Add";
export {Mul} from "./Mul";
import Operator from "./Operator";

export default Operator;
```

Möglich ist der Reexport des **gebündelten Imports**

```
//RepackedCalculator.ts
export * from "./Operators";
```

Beim erneuten Exportieren von Bundles können Deklarationen überschrieben werden, wenn sie ausdrücklich deklariert werden.

```
//FixedCalculator.ts
export * from "./Calculator"
import Operator from "./Calculator";
export class Add implements Operator {
    eval(a: number, b: number): number {
```

```
        return 42;
    }
}
```

Verwendungsbeispiel

```
//run.ts
import {Add, Mul} from "./FixedCalculator";

const add = new Add();
const mul = new Mul();

console.log(add.eval(1, 1)); // 42
console.log(mul.eval(3, 4)); // 12
```

Module - Exportieren und Importieren online lesen:
https://riptutorial.com/de/typescript/topic/9054/module---exportieren-und-importieren

# Kapitel 15: Schnittstellen

## Einführung

Eine Schnittstelle gibt eine Liste von Feldern und Funktionen an, die in jeder Klasse erwartet werden können, die die Schnittstelle implementiert. Umgekehrt kann eine Klasse eine Schnittstelle nicht implementieren, sofern nicht jedes Feld und jede Funktion auf der Schnittstelle angegeben ist.

Der Hauptvorteil der Verwendung von Schnittstellen besteht darin, dass Objekte unterschiedlicher Art auf polymorphe Weise verwendet werden können. Dies liegt daran, dass jede Klasse, die die Schnittstelle implementiert, mindestens diese Felder und Funktionen aufweist.

## Syntax

- Schnittstelle Schnittstellenname {
- Parametername: Parametertyp;
- optionalParameterName ?: parameterType;
- }

## Bemerkungen

# Schnittstellen vs. Typ-Aliase

Schnittstellen sind gut geeignet, um die Form eines Objekts anzugeben, z. B. für ein Personenobjekt, das Sie angeben könnten

```
interface person {
    id?: number;
    name: string;
    age: number;
}
```

Was aber, wenn Sie die Art und Weise darstellen möchten, wie eine Person in einer SQL-Datenbank gespeichert ist? Da jeder DB-Eintrag aus einer Zeile mit einer Form `[string, string, number]` (also einem Array von Strings oder Zahlen) besteht, können Sie dies nicht als Objekt-Shape darstellen, da die Zeile keine *Eigenschaften hat* als solches ist es nur ein Array.

Dies ist eine Gelegenheit, bei der Typen nützlich sind. Anstatt in jeder Funktion anzugeben, die eine `function processRow(row: [string, string, number])` akzeptiert, können Sie einen separaten Typalias für eine Zeile erstellen und diesen dann in jeder Funktion verwenden:

```
type Row = [string, string, number];
function processRow(row: Row)
```

# Offizielle Schnittstellendokumentation

https://www.typescriptlang.org/docs/handbook/interfaces.html

## Examples

### Fügen Sie einer vorhandenen Schnittstelle Funktionen oder Eigenschaften hinzu

Nehmen wir an, wir haben einen Verweis auf die `JQuery` Typdefinition und möchten diese erweitern, um zusätzliche Funktionen von einem Plugin zu haben, das wir hinzugefügt haben und für das keine offizielle Typdefinition vorhanden ist. Wir können es einfach erweitern, indem Sie Funktionen deklarieren, die durch Plugin in einer separaten Schnittstellendeklaration mit demselben `JQuery` Namen hinzugefügt wurden:

```
interface JQuery {
  pluginFunctionThatDoesNothing(): void;

  // create chainable function
  manipulateDOM(HTMLElement): JQuery;
}
```

Der Compiler führt alle Deklarationen mit demselben Namen zu einer zusammen. Weitere Informationen finden Sie unter Zusammenführen von Deklarationen .

### Klassenschnittstelle

Deklarieren Sie `public` Variablen und Methodentypen in der Schnittstelle, um festzulegen, wie anderer Typoscript-Code damit interagieren kann.

```
interface ISampleClassInterface {
  sampleVariable: string;

  sampleMethod(): void;

  optionalVariable?: string;
}
```

Hier erstellen wir eine Klasse, die die Schnittstelle implementiert.

```
class SampleClass implements ISampleClassInterface {
  public sampleVariable: string;
  private answerToLifeTheUniverseAndEverything: number;

  constructor() {
    this.sampleVariable = 'string value';
    this.answerToLifeTheUniverseAndEverything = 42;
  }
```

```
  public sampleMethod(): void {
    // do nothing
  }
  private answer(q: any): number {
    return this.answerToLifeTheUniverseAndEverything;
  }
}
```

Das Beispiel zeigt, wie Sie eine Schnittstelle `ISampleClassInterface` und eine Klasse `SampleClass` , `implements` die Schnittstelle `implements` .

## Schnittstelle erweitern

Angenommen, wir haben eine Schnittstelle:

```
interface IPerson {
    name: string;
    age: number;

    breath(): void;
}
```

Und wir wollen mehr spezifische Schnittstelle schaffen , die die gleichen Eigenschaften der Person hat, können wir es tun mit dem `extends` Stichwort:

```
interface IManager extends IPerson {
    managerId: number;

    managePeople(people: IPerson[]): void;
}
```

Zusätzlich können mehrere Schnittstellen erweitert werden.

## Verwenden von Schnittstellen zum Erzwingen von Typen

Typische Vorteile von Typescript sind die Erzwingung von Datentypen von Werten, die Sie im Code übergeben, um Fehler zu vermeiden.

Nehmen wir an, Sie machen eine Pet-Dating-Bewerbung.

Sie haben diese einfache Funktion, die prüft, ob zwei Haustiere miteinander kompatibel sind ...

```
checkCompatible(petOne, petTwo) {
  if (petOne.species === petTwo.species &&
      Math.abs(petOne.age - petTwo.age) <= 5) {
    return true;
  }
}
```

Dies ist vollständig funktionaler Code, aber es wäre für jemanden viel zu einfach, vor allem für andere Personen, die an dieser Anwendung arbeiten, die diese Funktion nicht geschrieben haben, dass sie nicht wissen, dass sie Objekte mit 'Spezies' und 'Alter' übergeben sollen. Eigenschaften.

Sie versuchen möglicherweise fehlerhaft `checkCompatible(petOne.species, petTwo.species)` und müssen dann die Fehler herausfinden, die ausgelöst werden, wenn die Funktion auf petOne.species.species oder petOne.species.age zugreift.

Eine Möglichkeit, dies zu verhindern, besteht darin, die gewünschten Eigenschaften für die Tierparameter festzulegen:

```
checkCompatible(petOne: {species: string, age: number}, petTwo: {species: string, age:
number}) {
    //...
}
```

In diesem Fall stellt Typescript sicher, dass alles, was an die Funktion übergeben wird, Eigenschaften wie 'Spezies' und 'Alter' hat (es ist in Ordnung, wenn sie zusätzliche Eigenschaften haben). Dies ist jedoch eine etwas unhandliche Lösung, selbst wenn nur zwei Eigenschaften angegeben sind. Mit Schnittstellen gibt es einen besseren Weg!

Zuerst definieren wir unsere Schnittstelle:

```
interface Pet {
  species: string;
  age: number;
  //We can add more properties if we choose.
}
```

Jetzt müssen wir nur noch die Art unserer Parameter als unsere neue Schnittstelle angeben, wie so ...

```
checkCompatible(petOne: Pet, petTwo: Pet) {
  //...
}
```

... und Typescript stellen sicher, dass die an unsere Funktion übergebenen Parameter die in der Pet-Schnittstelle angegebenen Eigenschaften enthalten!

### Generische Schnittstellen

Schnittstellen können wie Klassen auch polymorphe Parameter (auch bekannt als Generics) erhalten.

## Generische Parameter für Schnittstellen deklarieren

```
interface IStatus<U> {
    code: U;
}

interface IEvents<T> {
    list: T[];
    emit(event: T): void;
    getAll(): T[];
```

```
}
```

Hier können Sie sehen, dass unsere beiden Schnittstellen einige generische Parameter, **T** und **U, haben** .

## Generische Schnittstellen implementieren

Wir erstellen eine einfache Klasse, um die Schnittstelle **IEvents** zu implementieren.

```
class State<T> implements IEvents<T> {

    list: T[];

    constructor() {
        this.list = [];
    }

    emit(event: T): void {
        this.list.push(event);
    }

    getAll(): T[] {
        return this.list;
    }

}
```

Lassen Sie uns einige Instanzen unserer **State-** Klasse erstellen.

In unserem Beispiel behandelt die `State` Klasse einen generischen Status mit `IStatus<T>` . Auf diese Weise wird die Schnittstelle `IEvent<T>` auch einen `IStatus<T>` .

```
const s = new State<IStatus<number>>();

// The 'code' property is expected to be a number, so:
s.emit({ code: 200 }); // works
s.emit({ code: '500' }); // type error

s.getAll().forEach(event => console.log(event.code));
```

Hier wird unsere `State` Klasse als `ISatus<number>` eingegeben.

---

```
const s2 = new State<IStatus<Code>>();

//We are able to emit code as the type Code
s2.emit({ code: { message: 'OK', status: 200 } });

s2.getAll().map(event => event.code).forEach(event => {
    console.log(event.message);
    console.log(event.status);
});
```

Unsere `State` Klasse wird als `IStatus<Code>` eingegeben. Auf diese Weise können wir komplexere

Typen an unsere Emit-Methode übergeben.

Wie Sie sehen, können generische Schnittstellen ein sehr nützliches Werkzeug für statisch typisierten Code sein.

## Verwendung von Schnittstellen für Polymorphismus

Der Hauptgrund, Schnittstellen zu verwenden, um Polymorphie zu erreichen, und Entwicklern die Implementierung der Methoden auf eigene Weise in der Zukunft zu ermöglichen.

Angenommen, wir haben eine Schnittstelle und drei Klassen:

```
interface Connector{
    doConnect(): boolean;
}
```

Dies ist eine Konnektorschnittstelle. Jetzt werden wir das für die WLAN-Kommunikation implementieren.

```
export class WifiConnector implements Connector{

    public doConnect(): boolean{
        console.log("Connecting via wifi");
        console.log("Get password");
        console.log("Lease an IP for 24 hours");
        console.log("Connected");
        return true
    }

}
```

Hier haben wir unsere konkrete Klasse namens `WifiConnector` , die eine eigene Implementierung hat. Dies ist jetzt Typ `Connector` .

Jetzt erstellen wir unser `System` mit einem Komponenten- `Connector` . Dies wird als Abhängigkeitsinjektion bezeichnet.

```
export class System {
    constructor(private connector: Connector){ #inject Connector type
        connector.doConnect()
    }
}
```

`constructor(private connector: Connector)` Diese Linie ist hier sehr wichtig. `Connector` ist eine Schnittstelle und muss `doConnect()` . Als `Connector` ist eine Schnittstelle , dieses Klasse - `System` hat viel mehr Flexibilität. Wir können jeden Typ übergeben, der eine `Connector` Schnittstelle implementiert hat. In Zukunft erreicht der Entwickler mehr Flexibilität. Zum Beispiel möchte der Entwickler nun das Bluetooth-Verbindungsmodul hinzufügen:

```
export class BluetoothConnector implements Connector{
```

```
    public doConnect(): boolean{
        console.log("Connecting via Bluetooth");
        console.log("Pair with PIN");
        console.log("Connected");
        return true
    }

}
```

Stellen Sie sicher, dass Wifi und Bluetooth eine eigene Implementierung haben. Es gibt eine andere Art der Verbindung. Doch damit haben beide implementiert Typ `Connector` die sind jetzt Typ `Connector` . Damit wir diese als Konstruktorparameter an die `System` Klasse übergeben können. Dies wird als Polymorphismus bezeichnet. Der Klasse `System` ist jetzt nicht bekannt, ob es sich um Bluetooth / Wifi handelt, selbst wenn wir ein weiteres Kommunikationsmodul wie Inferade, Bluetooth5 und andere Module hinzufügen können, indem Sie lediglich die `Connector` Schnittstelle implementieren.

Dies wird als Duck-Typing bezeichnet . `Connector` ist jetzt dynamisch, da `doConnect()` nur ein Platzhalter ist und der Entwickler dies als seinen eigenen implementiert.

Wenn beim `constructor(private connector: WifiConnector)` wo `WifiConnector` eine konkrete Klasse ist, was wird passieren? Dann `System` - Klasse fest Paar nur mit WifiConnector nichts anderes. Hier löste das Interface unser Problem durch Polymorphismus.

## Implizite Implementierung und Objektform

TypeScript unterstützt Interfaces, der Compiler gibt jedoch JavaScript aus, was nicht der Fall ist. Daher gehen Schnittstellen beim Kompilieren effektiv verloren. Deshalb hängt die Typüberprüfung von Schnittstellen von der *Form* des Objekts ab - dh, ob das Objekt die Felder und Funktionen der Schnittstelle unterstützt - und nicht davon, ob die Schnittstelle tatsächlich implementiert ist oder nicht.

```
interface IKickable {
  kick(distance: number): void;
}
class Ball {
  kick(distance: number): void {
    console.log("Kicked", distance, "meters!");
  }
}
let kickable: IKickable = new Ball();
kickable.kick(40);
```

Auch wenn `Ball IKickable` nicht explizit implementiert, kann eine `Ball` Instanz einem `IKickable` zugewiesen (und als `IKickable` , selbst wenn der Typ angegeben wird.

Schnittstellen online lesen: https://riptutorial.com/de/typescript/topic/2023/schnittstellen

# Kapitel 16: So verwenden Sie eine Javascript-Bibliothek ohne Typendefinitionsdatei

## Einführung

Während einige vorhandene JavaScript - Bibliotheken haben Typ - Definitionsdateien , gibt es viele , die dies nicht tun.

TypeScript bietet einige Muster, um fehlende Deklarationen zu behandeln.

## Examples

### Deklarieren Sie eine beliebige globale

Es ist manchmal am einfachsten zu erklären nur einen globalen Typ `any` , vor allem in einfachen Projekten.

Wenn jQuery keine Typdeklarationen hatte ( es tut dies ), könnten Sie dies tun

```
declare var $: any;
```

Jetzt wird jeder Einsatz von $ eingegeben wird `any` .

### Erstellen Sie ein Modul, das einen Standardwert exportiert

Bei komplizierteren Projekten oder wenn Sie schrittweise eine Abhängigkeit eingeben möchten, kann es einfacher sein, ein Modul zu erstellen.

Verwenden Sie JQuery als Beispiel (obwohl es Typisierungen zur Verfügung hat ):

```
// place in jquery.d.ts
declare let $: any;
export default $;
```

Und dann können Sie diese Definition in einer beliebigen Datei in Ihrem Projekt importieren mit:

```
// some other .ts file
import $ from "jquery";
```

Nach diesem Import wird `$` als `any` eingegeben.

Wenn die Bibliothek über mehrere Variablen der obersten Ebene verfügt, exportieren und importieren Sie stattdessen nach Namen:

```
// place in jquery.d.ts
```

```
declare module "jquery" {
   let $: any;
   let jQuery: any;

   export { $ };
   export { jQuery };
}
```

Sie können dann beide Namen importieren und verwenden:

```
// some other .ts file
import {$, jQuery} from "jquery";

$.doThing();
jQuery.doOtherThing();
```

## Verwenden Sie ein Umgebungsmodul

Wenn Sie nur die *Absicht* eines Imports angeben möchten (also kein globales Objekt deklarieren möchten) und keine expliziten Definitionen verwenden möchten, können Sie ein Umgebungsmodul importieren.

```
// in a declarations file (like declarations.d.ts)
declare module "jquery";    // note that there are no defined exports
```

Sie können dann aus dem Umgebungsmodul importieren.

```
// some other .ts file
import {$, jQuery} from "jquery";
```

Alles, was aus dem deklarierten Modul (wie `$` und `jQuery` ) oben importiert wurde, hat den Typ `any`

So verwenden Sie eine Javascript-Bibliothek ohne Typendefinitionsdatei online lesen: https://riptutorial.com/de/typescript/topic/8249/so-verwenden-sie-eine-javascript-bibliothek-ohne-typendefinitionsdatei

# Kapitel 17: Strikte Nullprüfungen

## Examples

### Strikte Nullprüfungen in Aktion

Standardmäßig erlauben alle Typen in TypeScript `null` :

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // TypeScript does not complain, but this is a runtime error.
```

TypeScript 2.0 fügt Unterstützung für strikte Nullprüfungen hinzu. Wenn Sie `--strictNullChecks` wenn Sie `--strictNullChecks` `tsc` (oder dieses Flag in Ihrer `tsconfig.json` ), erlauben Typen nicht mehr `null` :

```
function getId(x: Element) {
  return x.id;
}
getId(null);  // error: Argument of type 'null' is not assignable to parameter of type
'Element'.
```

Sie müssen explizit `null` zulassen:

```
function getId(x: Element|null) {
  return x.id;  // error TS2531: Object is possibly 'null'.
}
getId(null);
```

Bei ordnungsgemäßer Überwachung wird der Codetyp überprüft und ordnungsgemäß ausgeführt:

```
function getId(x: Element|null) {
  if (x) {
    return x.id;  // In this branch, x's type is Element
  } else {
    return null;  // In this branch, x's type is null.
  }
}
getId(null);
```

### Nicht-Null-Zusicherungen

Der Nicht-Null-Assertionsoperator `!` ermöglicht Ihnen zu behaupten, dass ein Ausdruck nicht `null` oder `undefined` wenn der TypeScript-Compiler dies nicht automatisch ableiten kann:

```
type ListNode = { data: number; next?: ListNode; };

function addNext(node: ListNode) {
```

```
    if (node.next === undefined) {
        node.next = {data: 0};
    }
}

function setNextValue(node: ListNode, value: number) {
    addNext(node);

    // Even though we know `node.next` is defined because we just called `addNext`,
    // TypeScript isn't able to infer this in the line of code below:
    // node.next.data = value;

    // So, we can use the non-null assertion operator, !,
    // to assert that node.next isn't undefined and silence the compiler warning
    node.next!.data = value;
}
```

Strikte Nullprüfungen online lesen: https://riptutorial.com/de/typescript/topic/1727/strikte-nullprufungen

# Kapitel 18: tsconfig.json

## Syntax

- Verwendet das JSON-Dateiformat
- Kann auch Kommentare im JavaScript-Stil akzeptieren

## Bemerkungen

# Überblick

Das Vorhandensein einer Datei tsconfig.json in einem Verzeichnis gibt an, dass das Verzeichnis das Stammverzeichnis eines TypeScript-Projekts ist. Die Datei tsconfig.json gibt die Stammdateien und die Compileroptionen an, die zum Kompilieren des Projekts erforderlich sind.

# Tsconfig.json verwenden

- Durch Aufrufen von tsc ohne Eingabedateien sucht der Compiler in diesem Fall nach der Datei tsconfig.json, die im aktuellen Verzeichnis beginnt und die übergeordnete Verzeichniskette fortsetzt.
- Durch Aufruf von tsc ohne Eingabedateien und einer Befehlszeilenoption --project (oder nur -p), die den Pfad eines Verzeichnisses angibt, das eine Datei tsconfig.json enthält. Wenn Eingabedateien in der Befehlszeile angegeben werden, handelt es sich um tsconfig.json-Dateien

# Einzelheiten

Die Eigenschaft `"compilerOptions"` kann weggelassen werden. In diesem Fall werden die Standardwerte des Compilers verwendet. Siehe unsere vollständige Liste der unterstützten Compileroptionen .

Wenn in einer Datei `"files"` tsconfig.json `"files"` keine Eigenschaft `"files"` vorhanden ist, enthält der Compiler standardmäßig alle TypeScript-Dateien (* .ts oder * .tsx) im enthaltenden Verzeichnis und in den Unterverzeichnissen. Wenn eine Eigenschaft "files" vorhanden ist, sind nur die angegebenen Dateien enthalten.

Wenn die Eigenschaft `"exclude"` angegeben ist, enthält der Compiler alle TypeScript-Dateien (* .ts oder * .tsx) im übergeordneten Verzeichnis und den Unterverzeichnissen, mit Ausnahme der ausgeschlossenen Dateien oder Ordner.

Die Eigenschaft `"files"` kann nicht in Verbindung mit der Eigenschaft "exclude" verwendet werden. Wenn beide angegeben werden, hat die Eigenschaft "files" Vorrang.

Alle Dateien, auf die von den in der Eigenschaft `"files"` angegebenen Dateien verwiesen wird, sind ebenfalls enthalten. Wenn eine Datei B.ts von einer anderen Datei A.ts referenziert wird, kann B.ts ebenfalls nicht ausgeschlossen werden, es sei denn, die Referenzdatei A.ts ist ebenfalls in der Liste "Exclude" angegeben.

Eine `tsconfig.json` Datei darf vollständig leer sein, wodurch alle Dateien im `tsconfig.json` Verzeichnis und den Unterverzeichnissen mit den Standard-Compiler-Optionen kompiliert werden.

In der Befehlszeile angegebene Compiler-Optionen überschreiben die in der Datei tsconfig.json angegebenen.

# Schema

Schema finden Sie unter: http://json.schemastore.org/tsconfig

## Examples

### Erstellen Sie ein TypeScript-Projekt mit tsconfig.json

Das Vorhandensein einer Datei " **tsconfig.json** " gibt an, dass das aktuelle Verzeichnis das Stammverzeichnis eines TypeScript-fähigen Projekts ist.

Das Initialisieren eines TypeScript-Projekts oder besser die Datei tsconfig.json kann mit dem folgenden Befehl ausgeführt werden:

```
tsc --init
```

Ab TypeScript v2.3.0 und höher wird standardmäßig die folgende Datei tsconfig.json erstellt:

```
{
  "compilerOptions": {
    /* Basic Options */
    "target": "es5",                          /* Specify ECMAScript target version: 'ES3'
(default), 'ES5', 'ES2015', 'ES2016', 'ES2017', or 'ESNEXT'. */
    "module": "commonjs",                     /* Specify module code generation: 'commonjs',
'amd', 'system', 'umd' or 'es2015'. */
    // "lib": [],                             /* Specify library files to be included in the
compilation:  */
    // "allowJs": true,                       /* Allow javascript files to be compiled. */
    // "checkJs": true,                       /* Report errors in .js files. */
    // "jsx": "preserve",                     /* Specify JSX code generation: 'preserve',
'react-native', or 'react'. */
    // "declaration": true,                   /* Generates corresponding '.d.ts' file. */
    // "sourceMap": true,                     /* Generates corresponding '.map' file. */
    // "outFile": "./",                       /* Concatenate and emit output to single file.
*/
    // "outDir": "./",                        /* Redirect output structure to the directory.
*/
    // "rootDir": "./",                       /* Specify the root directory of input files.
Use to control the output directory structure with --outDir. */
    // "removeComments": true,                /* Do not emit comments to output. */
```

```
    // "noEmit": true,                        /* Do not emit outputs. */
    // "importHelpers": true,                 /* Import emit helpers from 'tslib'. */
    // "downlevelIteration": true,            /* Provide full support for iterables in 'for-
of', spread, and destructuring when targeting 'ES5' or 'ES3'. */
    // "isolatedModules": true,               /* Transpile each file as a separate module
(similar to 'ts.transpileModule'). */

    /* Strict Type-Checking Options */
    "strict": true                            /* Enable all strict type-checking options. */
    // "noImplicitAny": true,                 /* Raise error on expressions and declarations
with an implied 'any' type. */
    // "strictNullChecks": true,              /* Enable strict null checks. */
    // "noImplicitThis": true,                /* Raise error on 'this' expressions with an
implied 'any' type. */
    // "alwaysStrict": true,                  /* Parse in strict mode and emit "use strict"
for each source file. */

    /* Additional Checks */
    // "noUnusedLocals": true,                /* Report errors on unused locals. */
    // "noUnusedParameters": true,            /* Report errors on unused parameters. */
    // "noImplicitReturns": true,             /* Report error when not all code paths in
function return a value. */
    // "noFallthroughCasesInSwitch": true,    /* Report errors for fallthrough cases in switch
statement. */

    /* Module Resolution Options */
    // "moduleResolution": "node",            /* Specify module resolution strategy: 'node'
(Node.js) or 'classic' (TypeScript pre-1.6). */
    // "baseUrl": "./",                       /* Base directory to resolve non-absolute module
names. */
    // "paths": {},                           /* A series of entries which re-map imports to
lookup locations relative to the 'baseUrl'. */
    // "rootDirs": [],                        /* List of root folders whose combined content
represents the structure of the project at runtime. */
    // "typeRoots": [],                       /* List of folders to include type definitions
from. */
    // "types": [],                           /* Type declaration files to be included in
compilation. */
    // "allowSyntheticDefaultImports": true,  /* Allow default imports from modules with no
default export. This does not affect code emit, just typechecking. */

    /* Source Map Options */
    // "sourceRoot": "./",                    /* Specify the location where debugger should
locate TypeScript files instead of source locations. */
    // "mapRoot": "./",                       /* Specify the location where debugger should
locate map files instead of generated locations. */
    // "inlineSourceMap": true,               /* Emit a single file with source maps instead
of having a separate file. */
    // "inlineSources": true,                 /* Emit the source alongside the sourcemaps
within a single file; requires '--inlineSourceMap' or '--sourceMap' to be set. */

    /* Experimental Options */
    // "experimentalDecorators": true,        /* Enables experimental support for ES7
decorators. */
    // "emitDecoratorMetadata": true,         /* Enables experimental support for emitting
type metadata for decorators. */
  }
}
```

Die meisten, wenn nicht alle, Optionen werden automatisch generiert, wobei nur das Nötigste

unkommentiert bleibt.

Ältere Versionen von TypeScript, wie zum Beispiel v2.0.x und niedriger, würden eine tsconfig.json wie folgt generieren:

```
{
    "compilerOptions": {
        "module": "commonjs",
        "target": "es5",
        "noImplicitAny": false,
        "sourceMap": false
    }
}
```

## compileOnSave

Das Festlegen einer Eigenschaft der obersten Ebene `compileOnSave` signalisiert der IDE, dass alle Dateien für eine bestimmte **Datei tsconfig.json** beim Speichern **generiert** werden.

```
{
    "compileOnSave": true,
    "compilerOptions": {
        ...
    },
    "exclude": [
        ...
    ]
}
```

Diese Funktion ist seit TypeScript 1.8.4 und höher verfügbar, muss jedoch direkt von IDEs unterstützt werden. Derzeit sind Beispiele für unterstützte IDEs:

- Visual Studio 2015 mit Update 3
- JetBrains WebStorm
- Atom mit Atom-Typoskript

## Bemerkungen

Eine Datei tsconfig.json kann sowohl Zeilen- als auch Blockkommentare enthalten, wobei dieselben Regeln wie bei ECMAScript verwendet werden.

```
//Leading comment
{
    "compilerOptions": {
        //this is a line comment
        "module": "commonjs", //eol line comment
        "target" /*inline block*/ : "es5",
        /* This is a
        block
        comment */
    }
}
/* trailing comment */
```

### Konfiguration für weniger Programmierfehler

Es gibt sehr gute Konfigurationen, um Typisierungen zu erzwingen und weitere hilfreiche Fehler zu erhalten, die standardmäßig nicht aktiviert sind.

```
{
  "compilerOptions": {

    "alwaysStrict": true, // Parse in strict mode and emit "use strict" for each source file.

    // If you have wrong casing in referenced files e.g. the filename is Global.ts and you
have a /// <reference path="global.ts" /> to reference this file, then this can cause to
unexpected errors. Visite: http://stackoverflow.com/questions/36628612/typescript-transpiler-
casing-issue
    "forceConsistentCasingInFileNames": true, // Disallow inconsistently-cased references to
the same file.

    // "allowUnreachableCode": false, // Do not report errors on unreachable code. (Default:
False)
    // "allowUnusedLabels": false, // Do not report errors on unused labels. (Default: False)

    "noFallthroughCasesInSwitch": true, // Report errors for fall through cases in switch
statement.
    "noImplicitReturns": true, // Report error when not all code paths in function return a
value.

    "noUnusedParameters": true, // Report errors on unused parameters.
    "noUnusedLocals": true, // Report errors on unused locals.

    "noImplicitAny": true, // Raise error on expressions and declarations with an implied
"any" type.
    "noImplicitThis": true, // Raise error on this expressions with an implied "any" type.

    "strictNullChecks": true, // The null and undefined values are not in the domain of every
type and are only assignable to themselves and any.

    // To enforce this rules, add this configuration.
    "noEmitOnError": true     // Do not emit outputs if any errors were reported.
  }
}
```

Nicht genug? Wenn Sie ein Hardcoder sind und mehr wollen, können Sie Ihre TypeScript-Dateien mit tslint prüfen, bevor Sie sie mit tsc kompilieren. Prüfen Sie, wie Sie Tslint für noch strengeren Code konfigurieren können .

### preserveConstEnums

Typescript unterstützt Costant-Aufzählungszeichen, die über `const enum` deklariert werden.

Dies ist normalerweise nur Syntaxzucker, da die Aufzählungszeichen in kompiliertem JavaScript eingebettet sind.

Zum Beispiel den folgenden Code

```
const enum Tristate {
```

```
    True,
    False,
    Unknown
}

var something = Tristate.True;
```

kompiliert zu

```
var something = 0;
```

Obwohl die Vorstelung profitieren von inlining, können Sie Aufzählungen lieber halten , auch wenn costant (dh: Sie Lesbarkeit auf Entwicklungscode wünschen kann), dies zu tun , Sie setzen in haben **tsconfig.json** die `preserveConstEnums` in die clausole `compilerOptions` zu `true` .

```
{
    "compilerOptions": {
        "preserveConstEnums" = true,
        ...
    },
    "exclude": [
        ...
    ]
}
```

Auf diese Weise würde das vorige Beispiel wie alle anderen Enumerationen kompiliert, wie im folgenden Snippet gezeigt.

```
var Tristate;
(function (Tristate) {
    Tristate[Tristate["True"] = 0] = "True";
    Tristate[Tristate["False"] = 1] = "False";
    Tristate[Tristate["Unknown"] = 2] = "Unknown";
})(Tristate || (Tristate = {}));

var something = Tristate.True
```

tsconfig.json online lesen: https://riptutorial.com/de/typescript/topic/4720/tsconfig-json

# Kapitel 19: TSLint - Sicherstellung der Codequalität und -konsistenz

## Einführung

TSLint führt eine statische Analyse des Codes durch und erkennt Fehler und mögliche Probleme im Code.

## Examples

### Grundlegendes tslint.json-Setup

Dies ist ein grundlegendes `tslint.json` Setup welches

- Verwendung verhindert `any`
- erfordert geschweifte Klammern für `if` / `else` / `for` / `do` / `while` Anweisungen
- erfordert Anführungszeichen ( `"` ) für Zeichenfolgen

```
{
    "rules": {
        "no-any": true,
        "curly": true,
        "quotemark": [true, "double"]
    }
}
```

### Konfiguration für weniger Programmierfehler

Dieses tslint.json-Beispiel enthält eine Reihe von Konfigurationen, um weitere Typisierungen zu erzwingen, häufig auftretende Fehler oder anderweitig verwirrende Konstrukte zu erkennen, die dazu neigen, Fehler zu erzeugen, und folgen den Codierrichtlinien für TypeScript-Mitwirkende .

Um diese Regeln durchzusetzen, schließen Sie tslint in Ihren Buildprozess ein und überprüfen Sie Ihren Code, bevor Sie ihn mit tsc kompilieren.

```
{
  "rules": {
     // TypeScript Specific
     "member-access": true, // Requires explicit visibility declarations for class members.
     "no-any": true, // Diallows usages of any as a type declaration.
     // Functionality
     "label-position": true, // Only allows labels in sensible locations.
     "no-bitwise": true, // Disallows bitwise operators.
     "no-eval": true, // Disallows eval function invocations.
     "no-null-keyword": true, // Disallows use of the null keyword literal.
     "no-unsafe-finally": true, // Disallows control flow statements, such as return,
continue, break and throws in finally blocks.
     "no-var-keyword": true, // Disallows usage of the var keyword.
```

```
    "radix": true, // Requires the radix parameter to be specified when calling parseInt.
    "triple-equals": true, // Requires === and !== in place of == and !=.
    "use-isnan": true, // Enforces use of the isNaN() function to check for NaN references
instead of a comparison to the NaN constant.
    // Style
    "class-name": true, // Enforces PascalCased class and interface names.
    "interface-name": [ true, "never-prefix" ], // Requires interface names to begin with a
capital 'I'
    "no-angle-bracket-type-assertion": true, // Requires the use of as Type for type
assertions instead of <Type>.
    "one-variable-per-declaration": true, // Disallows multiple variable definitions in the
same declaration statement.
    "quotemark": [ true, "double", "avoid-escape" ], // Requires double quotes for string
literals.
    "semicolon": [ true, "always" ], // Enforces consistent semicolon usage at the end of
every statement.
    "variable-name": [true, "ban-keywords", "check-format", "allow-leading-underscore"] //
Checks variable names for various errors. Disallows the use of certain TypeScript keywords
(any, Number, number, String, string, Boolean, boolean, undefined) as variable or parameter.
Allows only camelCased or UPPER_CASED variable names. Allows underscores at the beginning
(only has an effect if "check-format" specified).
  }
}
```

## Standardmäßig einen vordefinierten Regelsatz verwenden

`tslint` kann einen vorhandenen Regelsatz erweitern und wird mit den `tslint:recommended` und `tslint:latest` .

> `tslint:recommended` ist ein stabiler, etwas meinungsstarker Satz von Regeln, den wir für die allgemeine TypeScript-Programmierung `tslint:recommended` . Diese Konfiguration folgt Semver, so dass keine Änderungen in Nebenversionen oder Patch-Releases vorgenommen werden.
>
> `tslint:latest` erweitert tslint: empfohlen und wird fortlaufend aktualisiert, um die Konfiguration für die neuesten Regeln in jeder TSLint-Version zu enthalten. Die Verwendung dieser Konfiguration kann zu grundlegenden Änderungen in kleineren Releases führen, da neue Regeln aktiviert werden, die zu Flusenfehlern in Ihrem Code führen. Wenn TSLint einen Hauptversions-Bump erreicht, wird tslint: Recommended aktualisiert, um identisch mit tslint: latest zu sein.

Text & Tabellen und Quellcode predfined ruleset

So kann man einfach verwenden:

```
{
  "extends": "tslint:recommended"
}
```

eine sinnvolle Startkonfiguration haben.

Man kann dann Regeln aus dieser Voreinstellung über `rules` überschreiben, z. B. war es für Knotenentwickler sinnvoll, `no-console` auf `false` :

```
{
  "extends": "tslint:recommended",
  "rules": {
    "no-console": false
  }
}
```

## Installation und Einrichtung

Um den Befehl tslint run zu installieren

```
npm install -g tslint
```

Tslint wird über die Datei `tslint.json` konfiguriert. Um den Standard-Konfigurationsbefehl zu initialisieren

```
tslint --init
```

So prüfen Sie die Datei auf mögliche Fehler im Dateibetriebsbefehl

```
tslint filename.ts
```

## Sätze von TSLint-Regeln

- tslint-microsoft-contrib
- tslint-eslint-regeln
- Codelyzer

Yeoman Genearator unterstützt alle diese Presets und kann auch erweitert werden

- Generator-tslint

TSLint - Sicherstellung der Codequalität und -konsistenz online lesen: https://riptutorial.com/de/typescript/topic/7457/tslint---sicherstellung-der-codequalitat-und--konsistenz

# Kapitel 20: TypeScript mit AngularJS

## Parameter

| Name | Beschreibung |
|---|---|
| `controllerAs` | ist ein Aliasname, dem Variablen oder Funktionen zugewiesen werden können. @see: https://docs.angularjs.org/guide/directive |
| `$inject` | Abhängigkeitsinjektionsliste, wird durch Winkelung und Übergabe als Argument an Konstruktorfunktionen gelöst. |

## Bemerkungen

Denken Sie bei der Ausführung der Anweisung in TypeScript daran, dass Sie diese Sprache mit benutzerdefiniertem Typ und Benutzeroberflächen erstellen können. Dies ist äußerst hilfreich bei der Entwicklung riesiger Anwendungen. Die von vielen IDE unterstützte Code-Vervollständigung zeigt Ihnen den möglichen Wert anhand des entsprechenden Typs an, mit dem Sie arbeiten. Es gibt also weit mehr, was Sie beachten sollten (im Vergleich zu VanillaJS).

"Code gegen Schnittstellen, keine Implementierungen"

## Examples

### Richtlinie

```
interface IMyDirectiveController {
    // specify exposed controller methods and properties here
    getUrl(): string;
}

class MyDirectiveController implements IMyDirectiveController {

    // Inner injections, per each directive
    public static $inject = ["$location", "toaster"];

    constructor(private $location: ng.ILocationService, private toaster: any) {
        // $location and toaster are now properties of the controller
    }

    public getUrl(): string {
        return this.$location.url(); // utilize $location to retrieve the URL
    }
}

/*
 * Outer injections, for run once controll.
 * For example we have all templates in one value, and we wan't to use it.
 */
```

```
export function myDirective(templatesUrl: ITemplates): ng.IDirective {
    return {
        controller: MyDirectiveController,
        controllerAs: "vm",

        link: (scope: ng.IScope,
               element: ng.IAugmentedJQuery,
               attributes: ng.IAttributes,
               controller: IMyDirectiveController): void => {

            let url = controller.getUrl();
            element.text("Current URL: " + url);

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

myDirective.$inject = [
    Templates.prototype.slug,
];

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
    directive(myDirective.prototype.slug, myDirective);
```

## Einfaches Beispiel

```
export function myDirective($location: ng.ILocationService): ng.IDirective {
    return {

        link: (scope: ng.IScope,
            element: ng.IAugmentedJQuery,
            attributes: ng.IAttributes): void => {

            element.text("Current URL: " + $location.url());

        },

        replace: true,
        require: "ngModel",
        restrict: "A",
        templateUrl: templatesUrl.myDirective,
    };
}

// Using slug naming across the projects simplifies change of the directive name
myDirective.prototype.slug = "myDirective";

// You can place this in some bootstrap file, or have them at the same file
angular.module("myApp").
```

```
    directive(myDirective.prototype.slug, [
        Templates.prototype.slug,
        myDirective
    ]);
```

## Komponente

Für einen einfacheren Übergang zu Angular 2 wird empfohlen, `Component` zu verwenden, das seit Angular 1.5.8 verfügbar ist

### myModule.ts

```
import { MyModuleComponent } from "./components/myModuleComponent";
import { MyModuleService } from "./services/MyModuleService";

angular
    .module("myModule", [])
    .component("myModuleComponent", new MyModuleComponent())
    .service("myModuleService", MyModuleService);
```

### components / myModuleComponent.ts

```
import IComponentOptions = angular.IComponentOptions;
import IControllerConstructor = angular.IControllerConstructor;
import Injectable = angular.Injectable;
import { MyModuleController } from "../controller/MyModuleController";

export class MyModuleComponent implements IComponentOptions {
    public templateUrl: string = "./app/myModule/templates/myComponentTemplate.html";
    public controller: Injectable<IControllerConstructor> = MyModuleController;
    public bindings: {[boundProperty: string]: string} = {};
}
```

### templates / myModuleComponent.html

```
<div class="my-module-component">
    {{$ctrl.someContent}}
</div>
```

### Controller / MyModuleController.ts

```
import IController = angular.IController;
import { MyModuleService } from "../services/MyModuleService";

export class MyModuleController implements IController {
    public static readonly $inject: string[] = ["$element", "myModuleService"];
    public someContent: string = "Hello World";

    constructor($element: JQuery, private myModuleService: MyModuleService) {
        console.log("element", $element);
    }

    public doSomething(): void {
        // implementation..
```

```
    }
}
```

### services / MyModuleService.ts

```
export class MyModuleService {
    public static readonly $inject: string[] = [];

    constructor() {
    }

    public doSomething(): void {
        // do something
    }
}
```

### irgendwo.html

```
<my-module-component></my-module-component>
```

TypeScript mit AngularJS online lesen: https://riptutorial.com/de/typescript/topic/6569/typescript-mit-angularjs

# Kapitel 21: TypeScript mit SystemJS

## Examples

### Hallo Welt im Browser mit SystemJS

#### Installieren Sie systemjs und plugin-typescript

```
npm install systemjs
npm install plugin-typescript
```

HINWEIS: Dadurch wird der TypScript 2.0.0-Compiler installiert, der noch nicht veröffentlicht ist.

Für TypeScript 1.8 müssen Sie Plugin-Typescript 4.0.16 verwenden

#### Erstellen Sie eine `hello.ts` Datei

```
export function greeter(person: String) {
    return 'Hello, ' + person;
}
```

#### Erstellen `hello.html` Datei `hello.html`

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>

    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

#### Erstellen Sie `config.js` - SystemJS-Konfigurationsdatei

```
System.config({
    packages: {
        "plugin-typescript": {
            "main": "plugin.js"
        },
```

```
        "typescript": {
            "main": "lib/typescript.js",
            "meta": {
                "lib/typescript.js": {
                    "exports": "ts"
                }
            }
        }
    },
    map: {
        "plugin-typescript": "node_modules/plugin-typescript/lib/",
        /* NOTE: this is for npm 3 (node 6) */
        /* for npm 2, typescript path will be */
        /* node_modules/plugin-typescript/node_modules/typescript */
        "typescript": "node_modules/typescript/"
    },
    transpiler: "plugin-typescript",
    meta: {
        "./hello.ts": {
            format: "esm",
            loader: "plugin-typescript"
        }
    },
    typescriptOptions: {
        typeCheck: 'strict'
    }
});
```

HINWEIS: Wenn Sie nicht über die Typprüfung möchten, entfernen Sie `loader: "plugin-typescript"` und `typescriptOptions` von `config.js` . Beachten Sie auch, dass JavaScript-Code niemals überprüft wird, insbesondere Code im Tag `<script>` im HTML-Beispiel.

### Probier es aus

```
npm install live-server
./node_modules/.bin/live-server --open=hello.html
```

### Bauen Sie es für die Produktion auf

```
npm install systemjs-builder
```

Erstellen Sie die `build.js` Datei:

```
var Builder = require('systemjs-builder');
var builder = new Builder();
builder.loadConfig('./config.js').then(function() {
    builder.bundle('./hello.ts', './hello.js', {minify: true});
});
```

Erstelle hallo.js von hallo.ts

```
node build.js
```

### Verwenden Sie es in der Produktion

Laden Sie einfach hello.js vor der ersten Verwendung mit einem Skript-Tag

`hello-production.html` datei:

```
<!doctype html>
<html>
<head>
    <title>Hello World in TypeScript</title>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="config.js"></script>
    <script src="hello.js"></script>
    <script>
        window.addEventListener('load', function() {
            System.import('./hello.ts').then(function(hello) {
                document.body.innerHTML = hello.greeter('World');
            });
        });
    </script>


</head>
<body>
</body>
</html>
```

TypeScript mit SystemJS online lesen: https://riptutorial.com/de/typescript/topic/6664/typescript-mit-systemjs

# Kapitel 22: TypeScript mit Webpack verwenden

## Examples

### webpack.config.js

install loader `npm install --save-dev ts-loader source-map-loader`

**tsconfig.json**

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react" // if you want to use react jsx
  }
}
```

```
module.exports = {
    entry: "./src/index.ts",
    output: {
        filename: "./dist/bundle.js",
    },

    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-
loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },
    /*****************************
     *  If you want to use react *
     *****************************/


    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
```

```
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    // externals: {
    //     "react": "React",
    //     "react-dom": "ReactDOM"
    // },
};
```

TypeScript mit Webpack verwenden online lesen:
https://riptutorial.com/de/typescript/topic/2024/typescript-mit-webpack-verwenden

# Kapitel 23: TypeScript-Kerntypen

## Syntax

- let variableName: Variablentyp;
- function functionName (parameterName: Variablentyp, parameterWithDefault: Variablentyp = ParameterDefault, optionalParameter ?: Variablentyp, ... variardicParameter: Variablentyp []): ReturnType {/*...*/};

## Examples

### Boolean

Ein Boolescher Wert stellt den grundlegendsten Datentyp in TypeScript dar, um True / False-Werte zuzuweisen.

```
// set with initial value (either true or false)
let isTrue: boolean = true;

// defaults to 'undefined', when not explicitely set
let unsetBool: boolean;

// can also be set to 'null' as well
let nullableBool: boolean = null;
```

### Nummer

Wie bei JavaScript sind Zahlen Gleitkommawerte.

```
let pi: number = 3.14;           // base 10 decimal by default
let hexadecimal: number = 0xFF;  // 255 in decimal
```

ECMAScript 2015 erlaubt binär und oktal.

```
let binary: number = 0b10;   // 2 in decimal
let octal: number = 0o755;   // 493 in decimal
```

### String

Textdatentyp:

```
let singleQuotes: string = 'single';
let doubleQuotes: string = "double";
let templateString: string = `I am ${ singleQuotes }`; // I am single
```

### Array

Ein Array von Werten:

```
let threePigs: number[] = [1, 2, 3];
let genericStringArray: Array<string> = ['first', '2nd', '3rd'];
```

## Enum

Ein Typ, um einen Satz numerischer Werte zu benennen:

Zahlenwerte standardmäßig auf 0:

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
let bestDay: Day = Day.Saturday;
```

Legen Sie eine Standardstartnummer fest:

```
enum TenPlus { Ten = 10, Eleven, Twelve }
```

oder Werte zuweisen:

```
enum MyOddSet { Three = 3, Five = 5, Seven = 7, Nine = 9 }
```

## Irgendein

Wenn Sie sich bezüglich eines Typs `any` sicher sind, ist `any` verfügbar:

```
let anything: any = 'I am a string';
anything = 5; // but now I am the number 5
```

## Leere

Wenn Sie überhaupt keinen Typ haben, wird er normalerweise für Funktionen verwendet, die nichts zurückgeben:

```
function log(): void {
    console.log('I return nothing');
}
```

`void` types Kann nur `null` oder `undefined` .

## Tupel

Array-Typ mit bekannten und möglicherweise unterschiedlichen Typen:

```
let day: [number, string];
day = [0, 'Monday'];      // valid
day = ['zero', 'Monday']; // invalid: 'zero' is not numeric
console.log(day[0]); // 0
```

```
console.log(day[1]); // Monday

day[2] = 'Saturday'; // valid: [0, 'Saturday']
day[3] = false;      // invalid: must be union type of 'number | string'
```

## Typen in Funktionsargumente und Rückgabewert. Nummer

Beim Erstellen einer Funktion in TypeScript können Sie den Datentyp der Funktionsargumente und den Datentyp für den Rückgabewert angeben

Beispiel:

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

Hier bedeutet die Syntax `x: number, y: number` , dass die Funktion zwei Argumente `x` und `y` annehmen kann und diese nur aus Zahlen und `(...): number {` kann, bedeutet, dass der Rückgabewert nur eine Zahl sein kann

Verwendungszweck:

```
sum(84 + 76) // will be return 160
```

Hinweis:

Das kannst du nicht tun

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

oder

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

Es werden folgende Fehler angezeigt:

`error TS2322: Type 'string' is not assignable to type 'number'` und der `error TS2322: Type 'number' is not assignable to type 'string'` werden

## Typen in Funktionsargumente und Rückgabewert. String

Beispiel:

```
function hello(name: string): string {
    return `Hello ${name}!`;
}
```

Hier ist die Syntax `name: string` bedeutet , dass die Funktion eines annehmen `name` Argument und dieses Argument kann nur String sein und `(...): string {` bedeutet , dass der Rückgabewert nur eine Zeichenfolge sein kann

Verwendungszweck:

```
hello('StackOverflow Documentation') // will be return Hello StackOverflow Documentation!
```

## String-Literal-Typen

Mit String-Literal-Typen können Sie den genauen Wert angeben, den eine Zeichenfolge haben kann.

```
let myFavoritePet: "dog";
myFavoritePet = "dog";
```

Jede andere Zeichenfolge gibt einen Fehler aus.

```
// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";
```

Zusammen mit Typenaliasen und Union-Typen erhalten Sie ein enumeartiges Verhalten.

```
type Species = "cat" | "dog" | "bird";

function buyPet(pet: Species, name: string) : Pet { /*...*/ }

buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" |
"bird". Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");
```

String-Literal-Typen können verwendet werden, um Überladungen zu unterscheiden.

```
function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet { /*...*/ }

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)
```

Sie funktionieren gut für benutzerdefinierte Typenschutz.

```
interface Pet {
    species: Species;
    eat();
    sleep();
}
```

```
interface Cat extends Pet {
    species: "cat";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet){
    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)
        pet.eat();
        pet.sleep();
    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)
        pet.eat();
        pet.sing();
        pet.sleep();
    }
}
```

Vollständiger Beispielcode

```
let myFavoritePet: "dog";
myFavoritePet = "dog";

// Error: Type '"rock"' is not assignable to type '"dog"'.
// myFavoritePet = "rock";

type Species = "cat" | "dog" | "bird";

interface Pet {
    species: Species;
    name: string;
    eat();
    walk();
    sleep();
}

interface Cat extends Pet {
    species: "cat";
}

interface Dog extends Pet {
    species: "dog";
}

interface Bird extends Pet {
    species: "bird";
    sing();
}
```

```
// Error: Interface 'Rock' incorrectly extends interface 'Pet'. Types of property 'species'
are incompatible. Type '"rock"' is not assignable to type '"cat" | "dog" | "bird"'. Type
'"rock"' is not assignable to type '"bird"'.
// interface Rock extends Pet {
//      type: "rock";
// }

function buyPet(pet: Species, name: string) : Pet;
function buyPet(pet: "cat", name: string): Cat;
function buyPet(pet: "dog", name: string): Dog;
function buyPet(pet: "bird", name: string): Bird;
function buyPet(pet: Species, name: string) : Pet {
    if(pet === "cat") {
        return {
            species: "cat",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Cat;
    } else if(pet === "dog") {
        return {
            species: "dog",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }
        } as Dog;
    } else if(pet === "bird") {
        return {
            species: "bird",
            name: name,
            eat: function () {
                console.log(`${this.name} eats.`);
            }, walk: function () {
                console.log(`${this.name} walks.`);
            }, sleep: function () {
                console.log(`${this.name} sleeps.`);
            }, sing: function () {
                console.log(`${this.name} sings.`);
            }
        } as Bird;
    } else {
        throw `Sorry we don't have a ${pet}. Would you like to buy a dog?`;
    }
}

function petIsCat(pet: Pet): pet is Cat {
    return pet.species === "cat";
}

function petIsDog(pet: Pet): pet is Dog {
```

```
    return pet.species === "dog";
}

function petIsBird(pet: Pet): pet is Bird {
    return pet.species === "bird";
}

function playWithPet(pet: Pet) {
    console.log(`Hey ${pet.name}, let's play.`);

    if(petIsCat(pet)) {
        // pet is now from type Cat (pet: Cat)

        pet.eat();
        pet.sleep();

        // Error: Type '"bird"' is not assignable to type '"cat"'.
        // pet.type = "bird";

        // Error: Property 'sing' does not exist on type 'Cat'.
        // pet.sing();

    } else if(petIsDog(pet)) {
        // pet is now from type Dog (pet: Dog)

        pet.eat();
        pet.walk();
        pet.sleep();

    } else if(petIsBird(pet)) {
        // pet is now from type Bird (pet: Bird)

        pet.eat();
        pet.sing();
        pet.sleep();
    } else {
        throw "An unknown pet. Did you buy a rock?";
    }
}

let dog = buyPet(myFavoritePet /* "dog" as defined above */, "Rocky");
// dog is from type Dog (dog: Dog)

// Error: Argument of type '"rock"' is not assignable to parameter of type "'cat' | "dog" |
"bird". Type '"rock"' is not assignable to type '"bird"'.
// buyPet("rock", "Rocky");

playWithPet(dog);
// Output: Hey Rocky, let's play.
//         Rocky eats.
//         Rocky walks.
//         Rocky sleeps.
```

## Schnittarten

Ein Schnittpunkttyp kombiniert das Mitglied von zwei oder mehr Typen.

```
interface Knife {
    cut();
```

```
}

interface BottleOpener{
    openBottle();
}

interface Screwdriver{
    turnScrew();
}

type SwissArmyKnife = Knife & BottleOpener & Screwdriver;

function use(tool: SwissArmyKnife){
    console.log("I can do anything!");

    tool.cut();
    tool.openBottle();
    tool.turnScrew();
}
```

## const Enum

Ein const-Enum ist dasselbe wie ein normales Enum. Nur wird zur Kompilierzeit kein Objekt generiert. Stattdessen werden die Literalwerte ersetzt, wenn das const Enum verwendet wird.

```
// Typescript: A const Enum can be defined like a normal Enum (with start value, specifig
values, etc.)
const enum NinjaActivity {
    Espionage,
    Sabotage,
    Assassination
}

// Javascript: But nothing is generated

// Typescript: Except if you use it
let myFavoriteNinjaActivity = NinjaActivity.Espionage;
console.log(myFavoritePirateActivity); // 0

// Javascript: Then only the number of the value is compiled into the code
// var myFavoriteNinjaActivity = 0 /* Espionage */;
// console.log(myFavoritePirateActivity); // 0

// Typescript: The same for the other constant example
console.log(NinjaActivity["Sabotage"]); // 1

// Javascript: Just the number and in a comment the name of the value
// console.log(1 /* "Sabotage" */); // 1

// Typescript: But without the object none runtime access is possible
// Error: A const enum member can only be accessed using a string literal.
// console.log(NinjaActivity[myFavoriteNinjaActivity]);
```

Zum Vergleich ein normales Enum

```
// Typescript: A normal Enum
enum PirateActivity {
```

```
    Boarding,
    Drinking,
    Fencing
}

// Javascript: The Enum after the compiling
// var PirateActivity;
// (function (PirateActivity) {
//     PirateActivity[PirateActivity["Boarding"] = 0] = "Boarding";
//     PirateActivity[PirateActivity["Drinking"] = 1] = "Drinking";
//     PirateActivity[PirateActivity["Fencing"] = 2] = "Fencing";
// })(PirateActivity || (PirateActivity = {}));

// Typescript: A normale use of this Enum
let myFavoritePirateActivity = PirateActivity.Boarding;
console.log(myFavoritePirateActivity); // 0

// Javascript: Looks quite similar in Javascript
// var myFavoritePirateActivity = PirateActivity.Boarding;
// console.log(myFavoritePirateActivity); // 0

// Typescript: And some other normale use
console.log(PirateActivity["Drinking"]); // 1

// Javascript: Looks quite similar in Javascript
// console.log(PirateActivity["Drinking"]); // 1

// Typescript: At runtime, you can access an normal enum
console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"

// Javascript: And it will be resolved at runtime
// console.log(PirateActivity[myFavoritePirateActivity]); // "Boarding"
```

TypeScript-Kerntypen online lesen: https://riptutorial.com/de/typescript/topic/2776/typescript-kerntypen

# Kapitel 24: Typische Skript-Beispiele

## Bemerkungen

Dies ist ein grundlegendes Beispiel, das eine generische Fahrzeugklasse erweitert und eine Fahrzeugbeschreibungsmethode definiert.

Weitere TypeScript-Beispiele finden Sie hier - TypeScript-Beispiele GitRepo

## Examples

### 1 einfaches Beispiel für die Vererbung von Klassen, das die Erweiterungen und das Super-Schlüsselwort verwendet

Eine generische Autoklasse verfügt über einige Fahrzeugeigenschaften und eine Beschreibungsmethode

```
class Car{
    name:string;
    engineCapacity:string;

    constructor(name:string,engineCapacity:string){
        this.name = name;
        this.engineCapacity = engineCapacity;
    }

    describeCar(){
        console.log(`${this.name} car comes with ${this.engineCapacity} displacement`);
    }
}

new Car("maruti ciaz","1500cc").describeCar();
```

HondaCar erweitert die bestehende generische Fahrzeugklasse und fügt neue Eigenschaften hinzu.

```
class HondaCar extends Car{
    seatingCapacity:number;

    constructor(name:string,engineCapacity:string,seatingCapacity:number){
        super(name,engineCapacity);
        this.seatingCapacity=seatingCapacity;
    }

    describeHondaCar(){
        super.describeCar();
        console.log(`this cars comes with seating capacity of ${this.seatingCapacity}`);
    }
}
new HondaCar("honda jazz","1200cc",4).describeHondaCar();
```

## Beispiel für eine statische Klassenvariable: Zählen Sie, wie viele Zeitmethoden aufgerufen werden

countInstance ist hier eine statische Klassenvariable

```
class StaticTest{
    static countInstance : number= 0;
    constructor(){
        StaticTest.countInstance++;
    }
}

new StaticTest();
new StaticTest();
console.log(StaticTest.countInstance);
```

Typische Skript-Beispiele online lesen: https://riptutorial.com/de/typescript/topic/7721/typische-skript-beispiele

# Kapitel 25: TypScript-Installations-Typoskript-und-Typ-Skript-Compiler-Tsc ausgeführt

## Einführung

So installieren Sie TypeScript und führen den TypeScript-Compiler für eine .ts-Datei über die Befehlszeile aus.

## Examples

### Schritte.

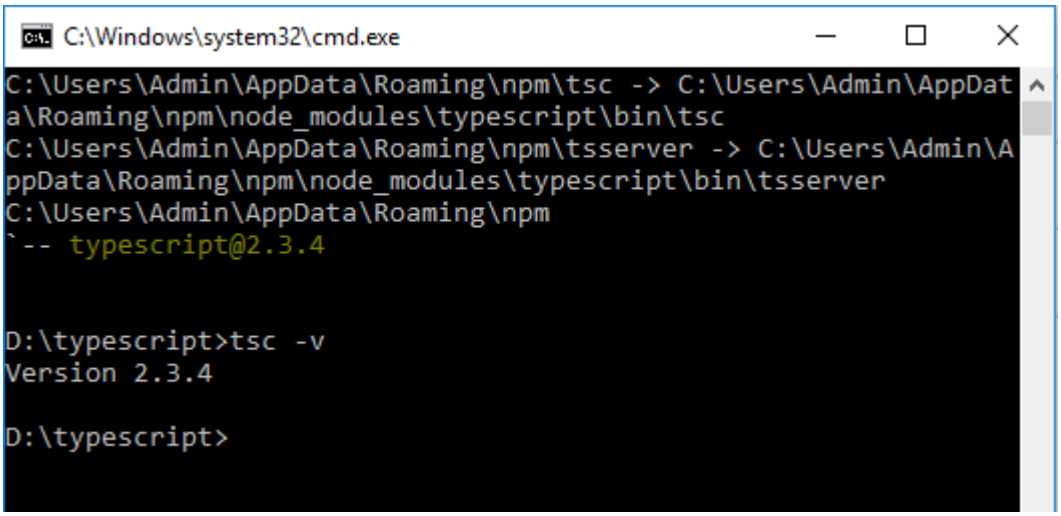
## Installation von Typescript und Ausführen des Typescript-Compilers.

#### So installieren Sie Typescript Comiler

```
npm install -g typescript
```

#### So überprüfen Sie die Typoskriptversion

```
tsc -v
```

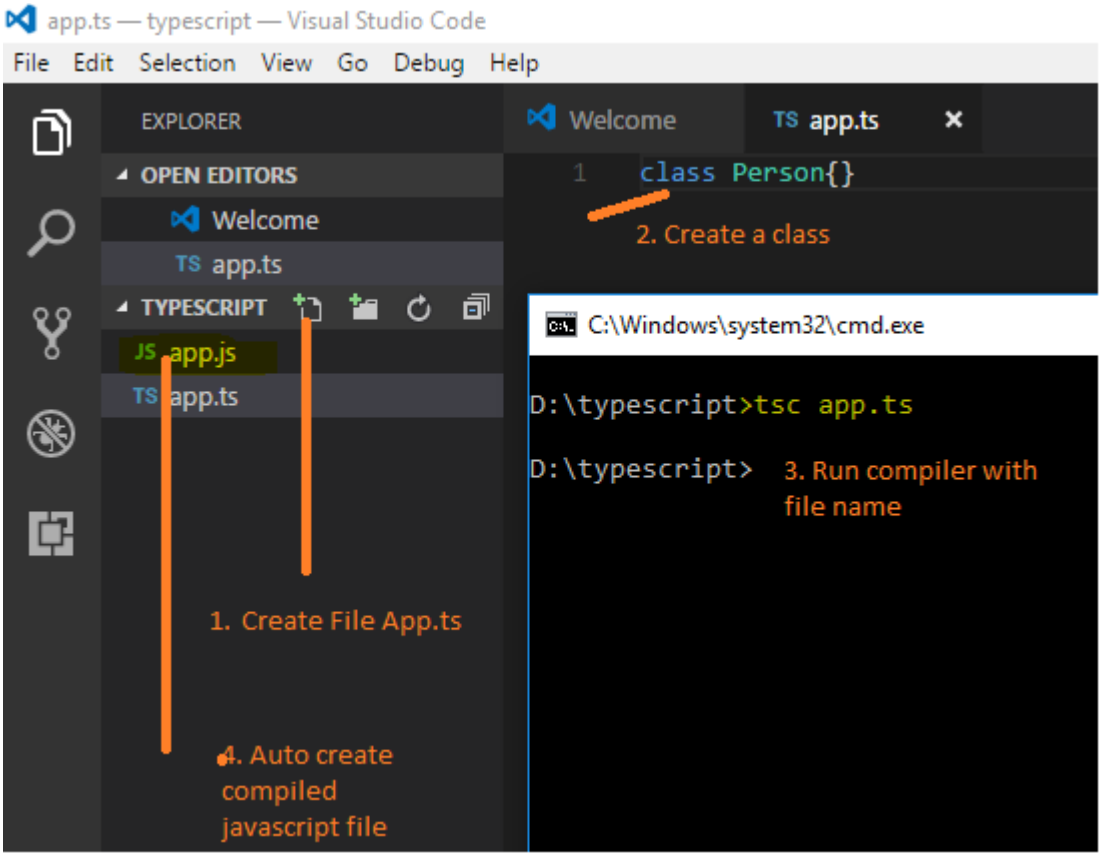



#### Laden Sie Visual Studio Code für Linux / Windows herunter

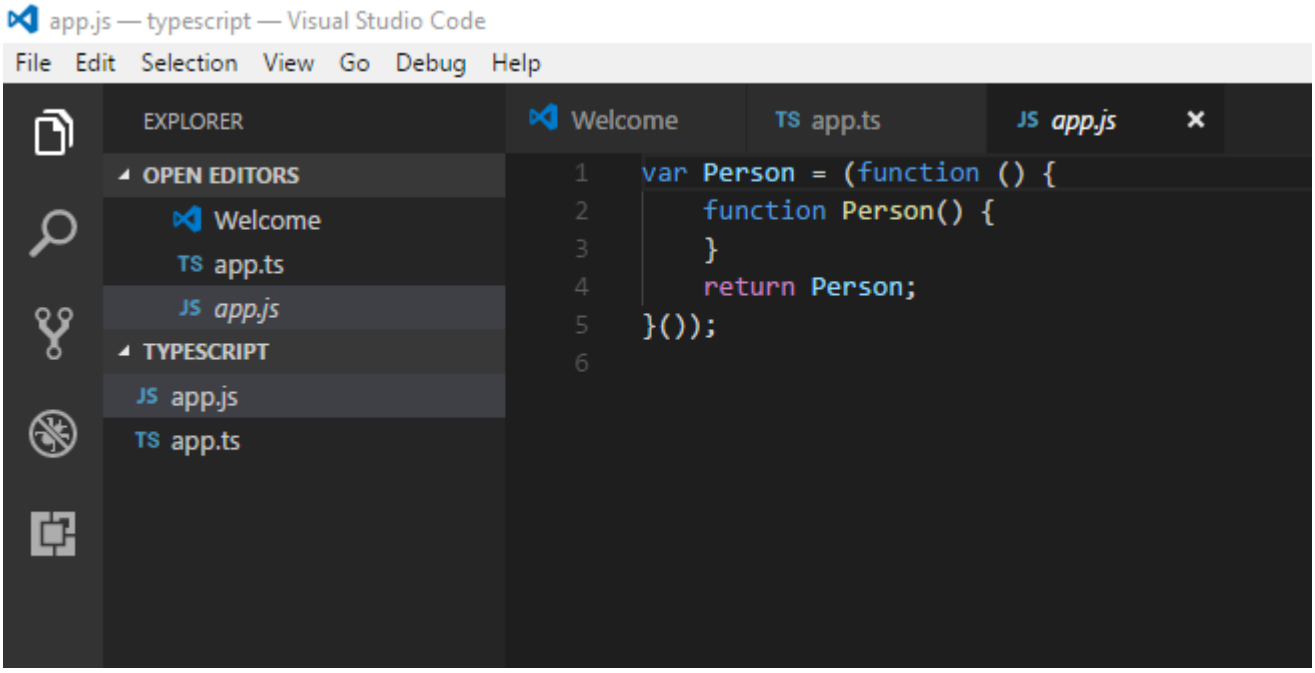
Visueller Code-Download-Link

1. Öffnen Sie Visual Studio Code

2. Öffnen Sie das gleiche Verzeichnis, in dem Sie den Typescript-Compiler installiert haben
3. Datei hinzufügen, indem Sie auf das Plus-Symbol im linken Bereich klicken
4. Erstellen Sie eine Grundklasse.
5. Kompilieren Sie Ihre Typ-Skript-Datei und generieren Sie eine Ausgabe.




Sehen Sie sich das Ergebnis in kompiliertem Javascript von geschriebenem Typoskriptcode an.




Vielen Dank.

TypScript-Installations-Typoskript-und-Typ-Skript-Compiler-Tsc ausgeführt online lesen:

https://riptutorial.com/de/typescript/topic/10503/typscript-installations-typoskript-und-typ-skript-compiler-tsc-ausgefuhrt

# Kapitel 26: Unit Testing

## Examples

### elsässisch

Alsatian ist ein in TypeScript geschriebenes Komponententest-Framework. Es ermöglicht die Verwendung von Testfällen und gibt TAP-konformes Markup aus.

Um es zu benutzen, installiere es von `npm` :

```
npm install alsatian --save-dev
```

Dann richten Sie eine Testdatei ein:

```
import { Expect, Test, TestCase } from "alsatian";
import { SomeModule } from "../src/some-module";

export SomeModuleTests {

    @Test()
    public statusShouldBeTrueByDefault() {
        let instance = new SomeModule();

        Expect(instance.status).toBe(true);
    }

    @Test("Name should be null by default")
    public nameShouldBeNullByDefault() {
        let instance = new SomeModule();

        Expect(instance.name).toBe(null);
    }

    @TestCase("first name")
    @TestCase("apples")
    public shouldSetNameCorrectly(name: string) {
        let instance = new SomeModule();

        instance.setName(name);

        Expect(instance.name).toBe(name);
    }

}
```

Eine vollständige Dokumentation finden Sie im GitHub-Repo von Alsatian .

### Chai-unveränderliches Plugin

1. Installation von npm chai, chai-immutable und ts-node

```
npm install --save-dev chai chai-immutable ts-node
```

2. Installieren Sie Typen für Mokka und Chai

```
npm install --save-dev @types/mocha @types/chai
```

3. Schreiben Sie eine einfache Testdatei:

```
  import {List, Set} from 'immutable';
  import * as chai from 'chai';
  import * as chaiImmutable from 'chai-immutable';

  chai.use(chaiImmutable);

  describe('chai immutable example', () => {
    it('example', () => {
      expect(Set.of(1,2,3)).to.not.be.empty;

      expect(Set.of(1,2,3)).to.include(2);
      expect(Set.of(1,2,3)).to.include(5);
    })
  })
```

4. Führen Sie es in der Konsole aus:

```
mocha --compilers ts:ts-node/register,tsx:ts-node/register 'test/**/*.spec.@(ts|tsx)'
```

## Band

Tape ist ein minimalistisches JavaScript - Testframework. Es gibt TAP-konformes Markup aus.

So installieren Sie ein `tape` mithilfe des Befehls " `npm` run"

```
npm install --save-dev tape @types/tape
```

Um `tape` mit Typescript zu verwenden, müssen Sie `ts-node` als globales Paket installieren, um diesen Ausführungsbefehl auszuführen

```
npm install -g ts-node
```

Nun können Sie Ihren ersten Test schreiben

```
//math.test.ts
import * as test from "tape";

test("Math test", (t) => {
    t.equal(4, 2 + 2);
    t.true(5 > 2 + 2);

    t.end();
});
```

Testlaufbefehl ausführen

```
ts-node node_modules/tape/bin/tape math.test.ts
```

In der Ausgabe sollten Sie sehen

```
TAP version 13
# Math test
ok 1 should be equal
ok 2 should be truthy

1..2
# tests 2
# pass  2

# ok
```

Gute Arbeit, Sie haben gerade Ihren TypeScript-Test durchgeführt.

**Führen Sie mehrere Testdateien aus**

Sie können mehrere Testdateien gleichzeitig mit Pfad-Platzhaltern ausführen. So führen Sie alle Typescript-Tests im `tests` Ausführungsbefehl aus

```
ts-node node_modules/tape/bin/tape tests/**/*.ts
```

## Scherz (ts-Scherz)

Jest ist ein schmerzloses JavaScript-Testframework von Facebook, mit ts-Jest kann TypeScript-Code getestet werden.

So installieren Sie Jest mit dem Befehl npm run

```
npm install --save-dev jest @types/jest ts-jest typescript
```

Um die Verwendung zu vereinfachen, installieren Sie `jest` als globales Paket

```
npm install -g jest
```

Damit `jest` mit TypeScript funktioniert, müssen Sie `package.json` eine Konfiguration `package.json`

```
//package.json
{
...
"jest": {
    "transform": {
      ".(ts|tsx)": "<rootDir>/node_modules/ts-jest/preprocessor.js"
    },
    "testRegex": "(/__tests__/.*|\\.(test|spec))\\.(ts|tsx|js)$",
    "moduleFileExtensions": ["ts", "tsx", "js"]
  }
```

```
}
```

Nun ist der `jest` fertig. Angenommen, wir haben Beispiel-Fizz-Buz zum Testen

```
//fizzBuzz.ts
export function fizzBuzz(n: number): string {
    let output = "";
    for (let i = 1; i <= n; i++) {
        if (i % 5 && i % 3) {
            output += i + ' ';
        }
        if (i % 3 === 0) {
            output += 'Fizz ';
        }
        if (i % 5 === 0) {
            output += 'Buzz ';
        }
    }
    return output;
}
```

Beispieltest könnte aussehen

```
//FizzBuzz.test.ts
/// <reference types="jest" />

import {fizzBuzz} from "./fizzBuzz";
test("FizzBuzz test", () =>{
    expect(fizzBuzz(2)).toBe("1 2 ");
    expect(fizzBuzz(3)).toBe("1 2 Fizz ");
});
```

Testlauf ausführen

```
jest
```

In der Ausgabe sollten Sie sehen

```
 PASS  ./fizzBuzz.test.ts
  ✓ FizzBuzz test (3ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.46s, estimated 2s
Ran all test suites.
```

# Codeabdeckung

`jest` unterstützt die Erzeugung von Berichten zur Codeabdeckung.

Um die Codeabdeckung mit TypeScript zu verwenden, müssen Sie `package.json` eine weitere

Konfigurationszeile `package.json` .

```
{
...
  "jest": {
  ...
    "testResultsProcessor": "<rootDir>/node_modules/ts-jest/coverageprocessor.js"
  }
}
```

Um Tests mit der Generierung des Abdeckungsberichts auszuführen

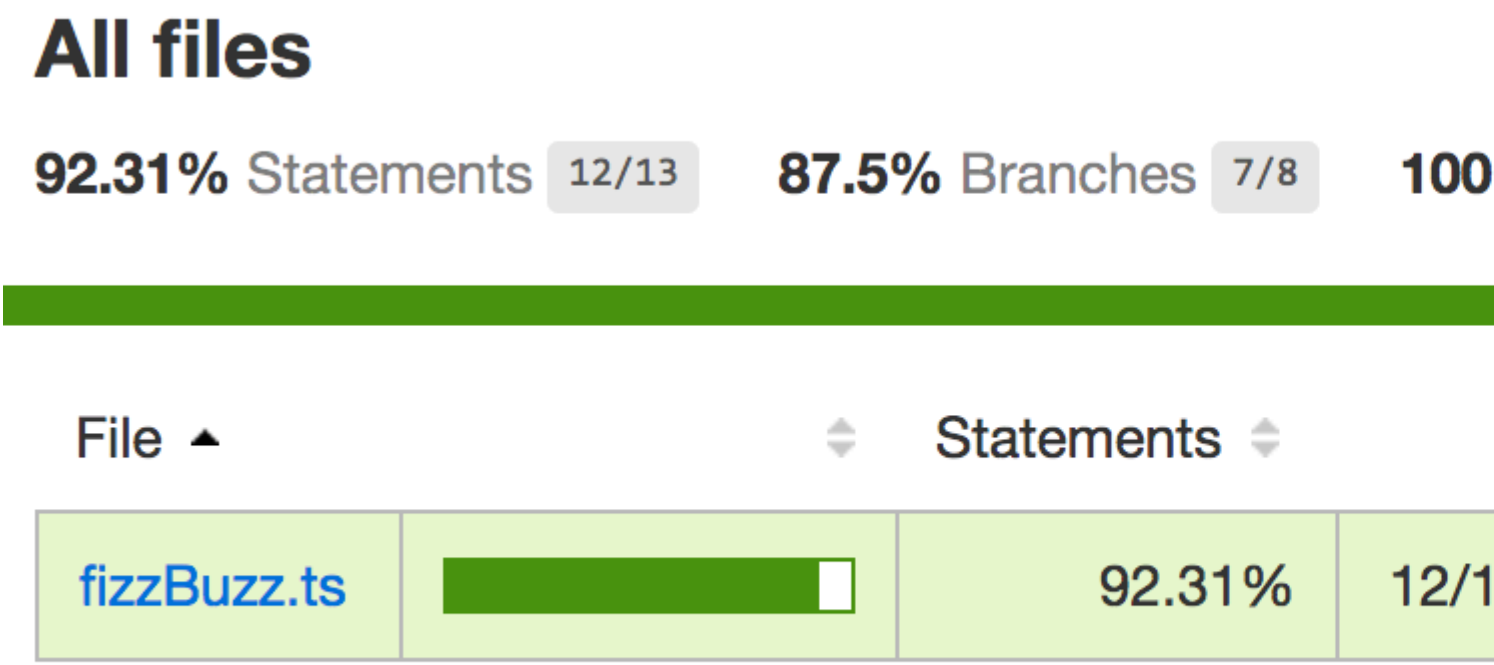
```
jest --coverage
```

Bei Verwendung mit unserem Beispiel-Fizz-Buzz sollten Sie es sehen

```
 PASS  ./fizzBuzz.test.ts
  ✓ FizzBuzz test (3ms)

----------|----------|----------|----------|----------|----------------|
File      | % Stmts | % Branch | % Funcs | % Lines |Uncovered Lines |
----------|----------|----------|----------|----------|----------------|
All files  |   92.31 |    87.5 |     100 |   91.67 |                |
 fizzBuzz.ts |   92.31 |    87.5 |     100 |   91.67 |             13 |
----------|----------|----------|----------|----------|----------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.857s
Ran all test suites.
```

`jest` erstellte außerdem Ordnerabdeckung `coverage` die Abdeckungsberichte in verschiedenen Formaten enthält, einschließlich benutzerfreundlicher HTML-Berichte in `coverage/lcov-report/index.html`




Unit Testing online lesen: https://riptutorial.com/de/typescript/topic/7456/unit-testing

# Kapitel 27: Veröffentlichen Sie TypeScript-Definitionsdateien

## Examples

### Definieren Sie die Definitionsdatei mit der Bibliothek auf npm

Fügen Sie Ihrem package.json Typisierungen hinzu

```
{
...
"typings": "path/file.d.ts"
...
}
```

Wenn nun diese Bibliothek importiert wird, lädt typescript die Typisierungsdatei

Veröffentlichen Sie TypeScript-Definitionsdateien online lesen:
https://riptutorial.com/de/typescript/topic/2931/veroffentlichen-sie-typescript-definitionsdateien

# Kapitel 28: Verwenden von Typescript mit React (JS & native)

## Examples

### ReactJS-Komponente in Typescript geschrieben

Sie können ReactJS-Komponenten problemlos in TypeScript verwenden. Benennen Sie einfach die Dateierweiterung 'jsx' in 'tsx' um:

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Damit Sie die Hauptfunktion von TypeScript (statische Typüberprüfung) voll ausnutzen können, müssen Sie einige Dinge tun:

**1) Konvertieren Sie React.createClass in eine ES6-Klasse:**

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Weitere Informationen zum Konvertieren nach ES6 finden Sie hier

**2) Requisiten und Statusschnittstellen hinzufügen:**

```
interface Props {
    name:string;
    optionalParam?:number;
}

interface State {
  //empty in our case
}

class HelloMessage extends React.Component<Props, State> {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
```

```
// TypeScript will allow you to create without the optional parameter
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
// But it does check if you pass in an optional parameter of the wrong type
ReactDOM.render(<HelloMessage name="Sebastian" optionalParam='foo' />, mountNode);
```

Jetzt zeigt TypeScript einen Fehler an, wenn der Programmierer vergisst, Requisiten zu übergeben. Oder wenn Sie versuchen, Requisiten zu übergeben, die nicht in der Schnittstelle definiert sind.

## Typoskript & Reagieren & Webpack

Globale Installation von Typoskript, Typisierung und Webpack

```
npm install -g typescript typings webpack
```

Loader installieren und Typoscript verknüpfen

```
npm install --save-dev ts-loader source-map-loader npm link typescript
```

> Durch die Verknüpfung von TypeScript kann ts-loader Ihre globale TypeScript-Installation verwenden, anstatt ein separates lokales CopyScript-Dokument zu benötigen

Installieren von `.d.ts` Dateien mit Typoskript 2.x

```
npm i @types/react --save-dev
npm i @types/react-dom --save-dev
```

Installieren von `.d.ts` Dateien mit `.d.ts` 1.x

```
typings install --global --save dt~react
typings install --global --save dt~react-dom
```

Konfigurationsdatei `tsconfig.json`

```
{
  "compilerOptions": {
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react"
  }
}
```

Konfigurationsdatei für `webpack.config.js`

```
module.exports = {
    entry: "<path to entry point>",// for example ./src/helloMessage.tsx
    output: {
        filename: "<path to bundle file>", // for example ./dist/bundle.js
    },
```

```
    // Enable sourcemaps for debugging webpack's output.
    devtool: "source-map",

    resolve: {
        // Add '.ts' and '.tsx' as resolvable extensions.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },

    module: {
        loaders: [
            // All files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'.
            {test: /\.tsx?$/, loader: "ts-loader"}
        ],

        preLoaders: [
            // All output '.js' files will have any sourcemaps re-processed by 'source-map-
loader'.
            {test: /\.js$/, loader: "source-map-loader"}
        ]
    },

    // When importing a module whose path matches one of the following, just
    // assume a corresponding global variable exists and use that instead.
    // This is important because it allows us to avoid bundling all of our
    // dependencies, which allows browsers to cache those libraries between builds.
    externals: {
        "react": "React",
        "react-dom": "ReactDOM"
    },
};
```

Endlich `webpack` oder `webpack -w` (für den Watch-Modus)

**Hinweis** : React und ReactDOM sind als extern gekennzeichnet

Verwenden von Typescript mit React (JS & native) online lesen: https://riptutorial.com/de/typescript/topic/1835/verwenden-von-typescript-mit-react--js--amp--native-

# Kapitel 29: Verwenden von Typescript mit RequireJS

## Einführung

RequireJS ist ein JavaScript-Datei- und -Modul-Loader. Es ist für die Verwendung im Browser optimiert, kann jedoch in anderen JavaScript-Umgebungen wie Rhino und Node verwendet werden. Die Verwendung eines modularen Skriptladers wie RequireJS verbessert die Geschwindigkeit und Qualität Ihres Codes.

Die Verwendung von TypeScript mit RequireJS erfordert die Konfiguration von tsconfig.json und das Einfügen eines Ausschnitts in eine beliebige HTML-Datei. Der Compiler konvertiert Importe von der Syntax von TypeScript in das RequireJS-Format.

## Examples

### HTML-Beispiel, bei dem zum Anhängen einer bereits kompilierten TypeScript-Datei eine requirJS-CDN verwendet wird.

```
<body onload="__init();">
    ...
    <script src="http://requirejs.org/docs/release/2.3.2/comments/require.js"></script>
    <script>
      function __init() {
        require(["view/index.js"]);
      }
    </script>
</body>
```

### tsconfig.json Beispiel zum Kompilieren, um Ordner mit dem erforderlichenJS-Importstil anzuzeigen.

```
{
  "module": "amd",    // Using AMD module code generator which works with requireJS
  "rootDir": "./src", // Change this to your source folder
  "outDir": "./view",
    ...
}
```

Verwenden von Typescript mit RequireJS online lesen:
https://riptutorial.com/de/typescript/topic/10773/verwenden-von-typescript-mit-requirejs

# Kapitel 30: Warum und wann TypScript verwenden?

## Einführung

Wenn Sie die Argumente für Typsysteme im Allgemeinen überzeugend finden, sind Sie mit TypeScript zufrieden.

Dies bringt viele der Vorteile eines Typsystems (Sicherheit, Lesbarkeit, verbesserte Werkzeuge) für das JavaScript-Ökosystem. Es hat auch einige Nachteile der Typsysteme (zusätzliche Komplexität und Unvollständigkeit).

## Bemerkungen

Die Vorteile von typisierten und nicht typisierten Sprachen werden seit Jahrzehnten diskutiert. Argumente für statische Typen sind:

1. Sicherheit: Mit Typsystemen können viele Fehler frühzeitig erkannt werden, ohne dass der Code ausgeführt wird. TypeScript kann so konfiguriert werden, dass weniger Programmierfehler auftreten
2. Lesbarkeit: Explizite Typen machen den Code für den Menschen verständlicher. Wie Fred Brooks schrieb : "Zeigen Sie mir Ihre Flussdiagramme und verbergen Sie Ihre Tische, und ich werde weiterhin verwirrt. Zeigen Sie mir Ihre Tabellen, und ich werde normalerweise keine Flussdiagramme benötigen;
3. Tooling: Typsysteme machen Computer für Computer verständlicher. Dadurch können Werkzeuge wie IDEs und Linters leistungsfähiger sein.
4. Leistung: Durch Typsysteme wird der Code schneller ausgeführt, da die Laufzeitprüfung nicht mehr erforderlich ist.

Da die Ausgabe von TypeScript unabhängig von ihrem Typ ist , hat TypeScript keinen Einfluss auf die Leistung. Das Argument für die Verwendung von TypeScript beruht auf den anderen drei Vorteilen.

Argumente gegen Typsysteme sind:

1. Komplexität hinzugefügt: Typsysteme können komplexer sein als die von ihnen beschriebene Sprachlaufzeit. Funktionen höherer Ordnung können leicht korrekt implementiert werden, sind aber schwer zu tippen . Durch den Umgang mit Typdefinitionen werden zusätzliche Hindernisse für die Verwendung externer Bibliotheken geschaffen.
2. Verbosity hinzugefügt: Typanmerkungen können Boilerplate zu Ihrem Code hinzufügen, wodurch die zugrunde liegende Logik schwerer zu befolgen ist.
3. Langsamere Iteration: Durch die Einführung eines Build-Schritts verlangsamt TypeScript den Bearbeitungs- / Speicherungs- / Neuladezyklus.
4. Unvollständigkeit: Ein Typsystem kann nicht gleichzeitig solide und vollständig sein. Es gibt

korrekte Programme, die TypeScript nicht zulässt. Und Programme, die TypeScript akzeptiert, können immer noch Fehler enthalten. Ein Typsystem macht das Testen nicht überflüssig. Wenn Sie TypeScript verwenden, müssen Sie möglicherweise länger warten, um die neuen ECMAScript-Sprachfunktionen zu verwenden.

TypeScript bietet einige Möglichkeiten, um alle diese Probleme zu beheben:

1. Komplexität hinzugefügt Wenn die Eingabe Teil eines Programms ist zu schwierig, Typoskript weitgehend kann unter Verwendung eines undurchsichtigen deaktiviert `any` Art. Gleiches gilt für externe Module.
2. Verbosität hinzugefügt. Dies kann teilweise durch Typ-Aliase und die Fähigkeit von TypeScript zum Ableiten von Typen behoben werden. Das Schreiben von klarem Code ist eine Kunst wie eine Wissenschaft: Entfernen Sie zu viele Typanmerkungen, und der Code ist möglicherweise für menschliche Leser nicht mehr klar.
3. Langsamere Iteration: Ein Build-Schritt ist in der modernen JS-Entwicklung relativ häufig und TypeScript ist bereits in die meisten Build-Tools integriert . Wenn TypeScript frühzeitig einen Fehler abfängt, können Sie einen gesamten Iterationszyklus einsparen!
4. Unvollständigkeit. Obwohl dieses Problem nicht vollständig gelöst werden kann, konnte TypeScript im Laufe der Zeit immer mehr ausdrucksstarke JavaScript-Muster erfassen. Zu den jüngsten Beispielen gehören das Hinzufügen von zugeordneten Typen in TypeScript 2.1 und Mixins in 2.2 .

Die Argumente für und gegen Typsysteme gelten im Allgemeinen auch für TypeScript. Die Verwendung von TypeScript erhöht den Aufwand für das Starten eines neuen Projekts. Aber im Laufe der Zeit, wenn das Projekt an Größe gewinnt und mehr Mitwirkende gewinnt, besteht die Hoffnung, dass die Profis (Sicherheit, Lesbarkeit, Werkzeug) des Einsatzes stärker werden und die Nachteile überwiegen. Dies spiegelt sich im Motto von TypeScript wider: "JavaScript skaliert."

# Examples

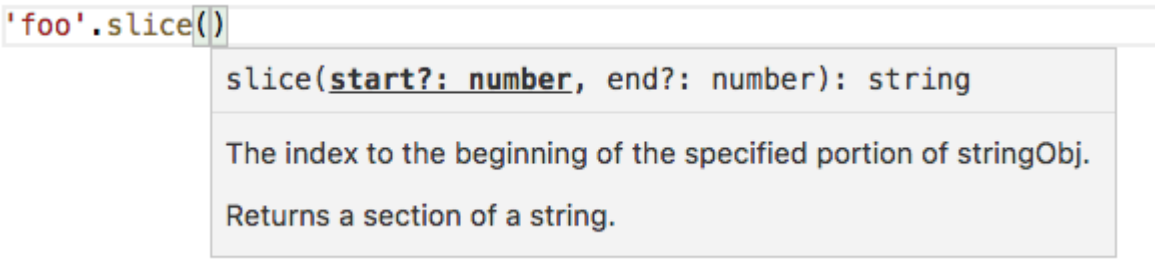
## Sicherheit

TypeScript fängt Typfehler früh durch statische Analyse ab:

```
function double(x: number): number {
  return 2 * x;
}
double('2');
//     ~~~ Argument of type '"2"' is not assignable to parameter of type 'number'.
```

## Lesbarkeit

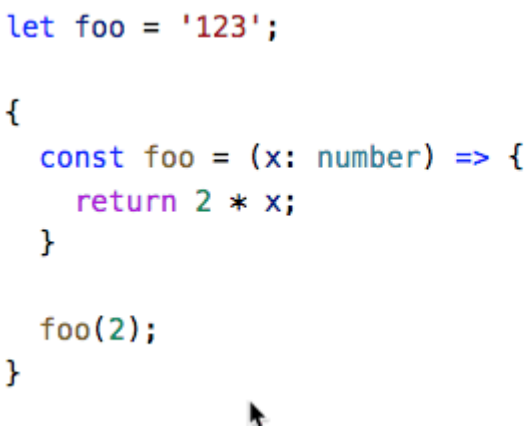
Mit TypeScript können Editoren kontextbezogene Dokumentationen bereitstellen:




Sie werden nie vergessen, ob `String.prototype.slice (start, stop)` oder `(start, length)` erneut dauert!

## Werkzeugbau

Mit TypeScript können Editoren automatisierte Refaktoren ausführen, die die Regeln der Sprachen kennen.




Hier zum Beispiel, Visual Studio - Code ist in der Lage Verweise auf die innere umbenennen `foo` ohne die äußere Veränderung `foo` . Dies wäre mit einem einfachen Suchen / Ersetzen schwierig.

Warum und wann TypScript verwenden? online lesen:
https://riptutorial.com/de/typescript/topic/9073/warum-und-wann-typscript-verwenden-

# Credits

| S. No | Kapitel | Contributors |
|---|---|---|
| 1 | Erste Schritte mit TypeScript | 2426021684, Alec Hansen, Blackus, BrunoLM, cdbajorin, ChanceM, Community, danvk, Florian Hämmerle, Fylax, goenning, islandman93, jengeb, Joshua Breeden, k0pernikus, Kiloku, KnottytOmo, Kuba Beránek, Lekhnath, Matt Lishman, Mikhail, mleko, RationalDev, Roy Dictus, Saiful Azad, Sam, samAlvin, Wasabi Fan, zigzag |
| 2 | Arrays | Udlei Nati |
| 3 | Aufzählungen | dimitrisli, Florian Hämmerle, Kevin Montrose, smnbbrv |
| 4 | Benutzerdefinierte Typenschutz | Kevin Montrose |
| 5 | Debuggen | Peopleware |
| 6 | Externe Bibliotheken importieren | 2426021684, Almond, artem, Blackus, Brutus, Dean Ward, dublicator, Harry, islandman93, JKillian, Joel Day, KnottytOmo, lefb766, Rajab Shakirov, Slava Shpitalny, tuvokki |
| 7 | Funktionen | br4d, hansmaad, islandman93, KnottytOmo, muetzerich, SilentLupin, Slava Shpitalny |
| 8 | Generics | danvk, hansmaad, KnottytOmo, Mathias Rodriguez, Muhammad Awais, Slava Shpitalny, Taytay |
| 9 | Integration mit Build-Tools | Alex Filatov, BrunoLM, Dan, dublicator, John Ruddell, mleko, Protectator, smnbbrv, void |
| 10 | Klasse Dekorateur | bruno, Remo H. Jansen, Stefan Rein |
| 11 | Klassen | adamboro, apricity, Cobus Kruger, Equiman, hansmaad, James Monger, Jeff Huijsmans, Justin Niles, KnottytOmo, Robin |
| 12 | Konfigurieren Sie das Typescript-Projekt, um alle Dateien in Typoscript zu kompilieren. | Rahul |
| 13 | Mixins | Fenton |
| 14 | Module - Exportieren | mleko |

| | | |
|---|---|---|
| | und Importieren | |
| 15 | Schnittstellen | ABabin, Aminadav, Aron, artem, Cobus Kruger, Fabian Lauer, islandman93, Joshua Breeden, Paul Boutes, Robin, Saiful Azad, Slava Shpitalny, Sunnyok |
| 16 | So verwenden Sie eine Javascript-Bibliothek ohne Typendefinitionsdatei | Bruno Krebs, Kevin Montrose |
| 17 | Strikte Nullprüfungen | bnieland, danvk, JKillian, Yaroslav Admin |
| 18 | tsconfig.json | bnieland, Fylax, goenning, Magu, Moriarty, user3893988 |
| 19 | TSLint - Sicherstellung der Codequalität und -konsistenz | Alex Filatov, James Monger, k0pernikus, Magu, mleko |
| 20 | TypeScript mit AngularJS | Chic, Roman M. Koss, Stefan Rein |
| 21 | TypeScript mit SystemJS | artem |
| 22 | TypeScript mit Webpack verwenden | BrunoLM, irakli khitarishvili, John Ruddell |
| 23 | TypeScript-Kerntypen | dublicator, Fenton, Fylax, Magu, Mikhail, Moriarty, RationalDev |
| 24 | Typische Skript-Beispiele | vashishth |
| 25 | TypScript-Installations-Typoskript-und-Typ-Skript-Compiler-Tsc ausgeführt | Rahul |
| 26 | Unit Testing | James Monger, leonidv, Louie Bertoncin, Matthew Harwood, mleko |
| 27 | Veröffentlichen Sie TypeScript-Definitionsdateien | 2426021684 |
| 28 | Verwenden von Typescript mit React | Aleh Kashnikau, irakli khitarishvili, islandman93, Rajab Shakirov , tBX |

| | | |
|---|---|---|
| | (JS & native) | |
| 29 | Verwenden von Typescript mit RequireJS | lilezek |
| 30 | Warum und wann TypScript verwenden? | danvk |