TYPESCRIPT

ppedv AG, Vadzim Naumchyk

INHALT

GETTING STARTED

TYPESCRIPT > IDEA

`#ts` `#typescript`

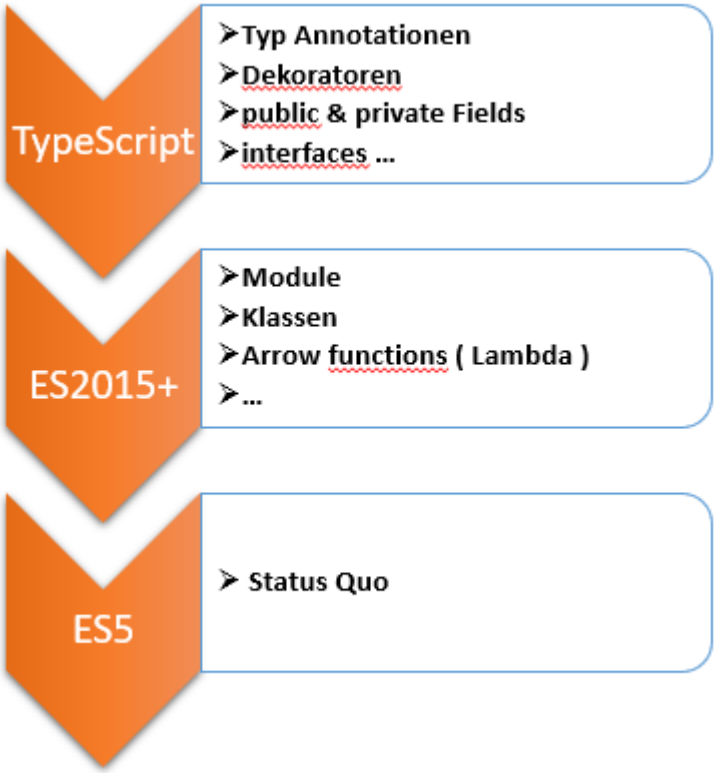
WAS IST TS

- eine Programmiersprache
- Obermenge von JavaScript
- entwickelt / maintained von Microsoft
- aktuelle Version 4.3.5 (01.06.2021)
    - `#checkForUpdates` https://github.com/microsoft/TypeScript/releases

WOZU IST TS

- um Fehler schon beim Kompilieren abzufangen (durch Typisierung und Code-Analyse-Tools)
- um Interfaces nutzen zu können
- um Code-Patterns besser umzusetzen
- um schneller zu programmieren
- um schneller zu debuggen

JS SUPERSET



STARTING LINKS

OFFIZIELLE QUELLEN

- HOMEPAGE https://www.typescriptlang.org/
- DOCS https://www.typescriptlang.org/docs/home.html
- CODE https://github.com/microsoft/TypeScript
- BLOG https://devblogs.microsoft.com/typescript/

STARTING TOOLS

`#tools`

- TypeScript Compiler tsc (um zu kompilieren: tsc yourfile.ts)

---

wenn **node.js und npm** genutzt werden, dann:

- nodejs & npmjs (um ts zu installieren: npm i -g typescript)
- Visual Studio Code (VSC)

- VSC Erweiterungen für TS / JS
  - TSLint / ESLint (muss für TS noch eingestellt werden)
  - JavaScript Snippets
- VSC Erweiterungen für HTML
  - open in browser

Wenn **MSBuild und Visual Studio** genutzt werden, dann:

- TypeScript als NuGet-Paket oder als VS-Erweiterung
- Visual Studio

Befehle aus der **VSCode-Command-Palette** (`ctr` + `shift` + `p`):

- TypeScript: Go To Project Configuration
- TypeScript: Select TypeScript Version
- TSLint: Manage workspace library execution

DEBUGGING

Mit Hilfe einer **map-Datei** sind Browser fähig, in ihren DevTools nicht die JS- sondern die TS-Datei zum Debuggen anzuzeigen.

Um die map-Datei erstellen zu lassen, muss die Option **'sourceMap'** in tsconfig auf 'true' gesetzt werden:

```
"sourceMap": true, /* Generates corresponding '.map' file. */
```

Weitere Infos:

- devtools in Firefox für TS https://hacks.mozilla.org/2019/09/debugging-typescript-in-firefox-devtools/

TS INSTALL / UPDATE

TypeScript muss global installiert werden.

```
npm install --global typescript@latest
```

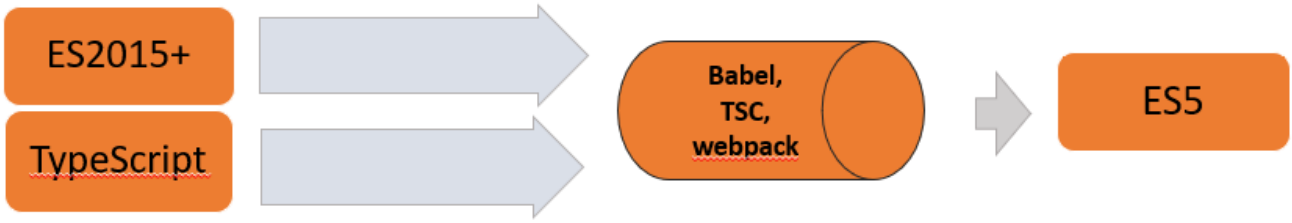
TS UPDATE

Aktualisiert wird TypeScript mit dem gleichen Befehl

```
npm install -g typescript@latest
```

COMPILING

An HTML-Dateien können **nur JS-Dateien** angeschlossen werden. Das heißt, um TS-Code auszutesten, braucht man seine kompilierte JS-Version.



Mit 'tsc --init' wird eine **TS-Config**-Datei angelegt.

Die Stelle, wo diese Datei angelegt wird, entscheidet, in welchen Ordnern der Compiler nach den TS-Dateien nachschauen soll.

TS USE

Wenn eine tsconfig-Datei erstellt wurde, dann gibt es 2 Möglichkeiten zu kompilieren:

1. **Alle Dateien** automatisch kompilieren lassen
2. Eine **Datei** kompilieren lassen

Option 1:

```
tsc -w
```

Das startet den watch-Modus.

Option 2:

```
tsc dateiname.ts
```

oder in der ts-Datei direkt: `strg shift b`. 'B' steht für 'build'.

TSC > IDEA

#tsc

WAS IST TSC

- ein CLI (command line interface) Programm / Tool
- tsc: TypeScript Compiler
- Version ist gleich wie vom npm-Paket 'typescript'

Was genau und wie der Compiler seine Arbeit **für ein TS-Projekt** erledigen soll, ist in der Datei **'tsconfig.json'** definiert.

Der Compiler hat aber im globalen Scope seine **Default-Einstellungen**.

Anwesenheit der tsconfig.json-Datei in einem Verzeichnis markiert es als **Root von einem TS-Projekt**.

Das heißt, beim Aufruf von tsc in watch-Modus werden die Dateien ab dem Verzeichnis beobachtet, wo tsconfig liegt, und nicht ab dem Verzeichnis, aus welchem tsc gestartet wurde.

TSCONFIG.JSON

Unter anderem wird hier definiert,

- in welche **JS-Version** kompiliert werden soll
- wo die **JS-Dateien** landen sollen
- ob **strict-Modus** bei JS angewendet werden soll

Bei der mit 'tsc --init' erstellten Datei steht ein **Kommentar bei jeder Konfiguration**.

Das erleichtert das Anpassen von Config-Optionen.

COMPILER CONFIGURATION

```
{
  "compilerOptions": {
    /* Visit https://aka.ms/tsconfig.json to read more about this file */

    /* Basic Options */
```

```
    // "incremental": true,                         /* Enable incremental
compilation */
    "target": "es5",                                /* Specify ECMAScript target
version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019',
'ES2020', 'ES2021', or 'ESNEXT'. */
    /* ... viele weitere Optionen */
    /* ... */
  }
}
```

TYPES

TYPES > INTRO

TYPISIERUNGSARTEN BEI SPRACHEN

- JS - **dynamische** Typisierung (Fehler treten erst beim Ausführen auf)
- TS - **statische** Typisierung (Fehler treten schon beim Compilieren auf)

Statisch-typisierte Sprachen verlangen die Typdeklarationen von Sprachkonstrukten schon bevor sie genutzt werden.

Datentypen unterstützen insbesondere die Entwicklungsumgebung:

- Auto-**Vervollständigung**
- **Fehlermeldungen** bei nicht passenden Datentypen (**Type-Checking**)

Beim build: TypeScript wird in JavaScript übersetzt, alle Typeninformationen gehen dabei verloren.

TYPE CHECKING

Datentypen werden von 'Type-Checking' **beim Deklarieren** von Variablen ermittelt.

Entweder durch die Angabe vom **Datentyp**:

```
let vorname: string;
// und später irgendwann:
vorname = 'Andreas';
```

Oder durch unmittelbare **Initialisierung**. Dann wird der Datentyp am Wert ermittelt:

```
let nachname = 'Krause'; // nachname ist ein string
```

Beides (Typangabe und Initialisierung) geht auch:

```
let alter: number = 40;
```

PRIMITIVES

Gleich wie in JS:

```
let completed: boolean = false;
let age: number = 32;
```

Obwohl TS strict typisiert ist, es gibt **keine** separaten Datentypen für **Ganz- und Gleitkommazahlen**

- boolean
- string
- number
- void
- never
- undefined
- null
- any (Top-Type, Universal Supertype, #v1.0+)
- unknown (noch ein Top-Type, #v3.0+)

Spezielle Typen wie void, never und unknown werden in späteren Kapiteln behandelt.

OVERLAPPING TYPES

Begriffe:

- overlapping types / Typen, die sich überschneiden
- compatible types / kompatible Typen
- subtype compatibility / Untertyp-Kompatibilität
- assignment compatibility / Zuweisung-Kompatibilität

SPECIAL TYPES

- union types
- tagged union types
- intersection types

- type aliases

Spezielle Datentypen werden im Kapitel 'More Types' behandelt.

ANY

Kompatibilität von any:

- any = [*primitive* | object]
- [*primitive* | object] = any

any: lässt alle Typen zu

```
let inputBox: any = document.querySelector('#inputbox');
```

TS & HTML

Für HTML-Elemente gibt es in JS **vordefinierte Objekte**. Z.B. mit dem Tag `<img>` wird ein HTMLImageElement angelegt.

In TS wird auf die **Struktur von solchen Objekten** strengst geachtet. Z.B. HTMLParagraphElement hat Eigenschaft 'textContent' aber nicht die Eigenschaft 'value'. Und bei HTMLInputElement umgekehrt: Eigenschaft 'value' ist drin, 'textContent' aber nicht.

Aber **nicht alle HTML-Tags** haben entsprechende Objekte. Z.B. für `<header>` gibt es kein HTMLHeaderElement, ein TS-Äquivalent zu Header ist ein höheres Objekt in der Prototypenkette, nämlich HTMLElement.

TS & DOM TRAVERSING

#traversing #DOM-quering

Es ist zu beachten, einige DOM-Traversing-Methoden (wie 'getElementById' als Beispiel) geben ein HTMLElement zurück. Also kein HTMLParagraphElement oder ein HTMLDivisionElement. Es wird ein allgemeineres Objekt 'HTMLElement' zurückgegeben.

Das liegt daran, dass die Methode 'getElementById()' anhand von der übergebenen ID nicht ermitteln kann, welche Art von einem HTML Element abgefangen wird. Um alle möglichen Varianten zu bedecken, wird die Rückgabe als Datentyp geliefert, von welchem alle genaueren HTML Elemente abgeleitet sind. Nämlich 'HTMLElement'-Typ.

Das gefundene Element kann man zu dem gewünschten Datentyp aber wie folgt anpassen:

```
// assert the return value from getElementById
const div:HTMLDivElement = document.getElementById('div_id') as HTMLDivElement;
```

UNION TYPE

Union Type kann als Datentyp bei Variablen eingesetzt werden, die **Werte von mehreren Datentypen** akzeptieren sollen

```
let figure: string | number | undefined
```

TYPE ALIASES

Wird ein Uniontype (oder ein anderer benutzerdefinierter Datentyp) **öfter verwendet**, kann dieser mit dem Schlüsselwort `type` angelegt werden:

```
type myStringNumberType = string | number
let figure: myStringNumberType
```

Schlüsselwort `type` für benutzerdefinierte Datentypen.

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
// ...
}
```

TYPE ASSERTION

'Assertion' zu Deutsch: 'Behauptung'.

Type-Assertion ist **kein Type-Casting**, es gibt keine Typ-Prüfung.

Der Programmierer 'behauptet', dass eine bestimmte Variable von einem bestimmten Typ ist.

Es wird dem Compiler gesagt: 'wir sind sicher, der Wert der Variable entspricht dem angegebenen Datentyp.'

Zwei Schreibweisen:

- `as` - as-syntax
- `<>` - angle-brackets syntax

```
// 1.
let someValue1: any = "this is a string";
let strLength1: number = (<string>someValue).length;

// 2.
let someValue2: any = "this is a string";
let strLength2: number = (someValue as string).length;
```

NULLABLE TYPES

Eine besondere Bedeutung haben die Unions mit den Optionen `null` oder `undefined`.

Die Methode 'getElementById()' liefert z.B. ein HTMLElement zurück, falls ein HTMLElement mit der angegebenen ID existiert, oder ein 'null', falls das Element nicht gefunden wurde.

Eine Variable, die die Rückgabe von dieser Methode abfängt, muss den entsprechenden Datentyp haben, nämlich 'HTMLElement | null'.

Wurde ein HTMLElement gefunden und man will weitere Operationen daran vornehmen, bekommt man die Fehlermeldung 'das Element ist möglicherweise ein null'.

- `!` - non-null assertion operator
- `?` - optional property access operator

NON NULL ASSERTION

Operator '!' löscht die **Option 'null' oder 'undefined'** bei einer Variable mit nullable Union.

Unoffiziell nennt man die Schreibweise mit dem '!'-Operator als Bang-Syntax.

```
let userName: string = (<HTMLInputElement>
document.getElementById('user')!).value;
```

OPTIONAL CHAINING

Operator '?' unterdrückt die Fehlermeldung von TS, dass ein Ausdruck eventuell 'null' oder 'undefined' ist.

```
let x = foo?.bar.baz();
```

Das entspricht dem Ausdruck

```
let x = foo === null || foo === undefined ? undefined : foo.bar.baz();
```

Das bedeutet, wenn foo null oder undefined ist, ein undefined wird rauskommen, sonst wird die Anweisung ausgeführt.

Der Operator ? nach dem foo, unterdrückt nicht die TS-Hinweise bezüglich 'bar'.

TYPE GUARDS

Typ-Wärter / Typ-Wächter

- typeof
- instanceof
- in
- benutzerdefinierter Typ-Wächter

USER DEFINED TYPE GUARD

Benutzerdefinierter Typ-Wächter ist eine Funktion.

Rückgabetyp dieser Funktion ist ein Typ-Prädikat.

FUNCTIONS

FUNCTIONS > INTRO

Typescript prüft, ob die **Anzahl von Parametern** der Anzahl der Argumente in der Funktionsdefinition entspricht.

```
function greeter(person){…}
greeter(); // error: expected 1 argument
```

RETURNING NO DATA - VOID

Funktionen **ohne return-Ausdruck** geben trotzdem einen Wert zurück.

Dieser Wert ist vom Datentyp **'void'**.

void ist kompatibel mit undefined und null.

```
function warnUser(): void {
  alert('this is a warning message');
}
```

ARGS & RETURN TYPES

Wir können Parametertypen und Rückgabetypen angeben

```
function repeatString(text: string, times: number): string {
  return ...;
}
```

OPT PARAMS

Optionale Parameter.

Eine Funktion kann **mit oder ohne optionale Parameter** aufgerufen werden.

Es ist nützlich, weil TS die Anzahl von übergebenen Argumenten überprüft.

```
function buildName(
    firstName: string, lastName?: string
): string
{
  return firstName + ' ' + lastName;
}
```

Optionale Parameter werden bei den Funktionsdefinitionen **nach den obligatorischen** Parametern angegeben.

FUNCTION CONSTRUCTOR

In TS kann man Funktionen auch mit dem **Konstruktor 'Function()'** anlegen:

```
let myFunction = new Function("a", "b", "return a * b");
let x = myFunction(4, 3);
console.log(x)
```

DATA STRUCTURES

ARRAY

Die meist verbreitete Struktur in Programmiersprachen ist ein Array.

Es gibt **zwei Schreibweisen** für Arrays als Datentyp in TypeScript:

- *itemDataType*[ ]
- Array<*itemDataType*>

z.B.:

```
// _itemDataType_[ ]
let names: string[]
names = ['Anna', 'Bernhardt', 'Caroline']

// Array<_itemDataType_>
let numbers: Array<number>
numbers = [34, 546.234, 143]

let mischmasch: Array<any>
mischmasch = [1, true, "text"]
```

Es gibt zwei Arten von Konstruktoren für Arrays, mit jeweils drei Overloads (Varianten)

Nicht generischer Konstruktor:

- Array()
- Array(*anzahlVonEinträgen*)
- Array(*einträge*)

z.B.

```
let names1 = new Array()
let names2 = new Array(4)
let names3 = new Array('Max', 'Maxim', 'Maximilian')
```

Generischer Konstruktor:

- Array<*datenTypVonEinträgen*>()
- Array<*datenTypVonEinträgen*>(*anzahlVonEinträgen*)

- Array<*datenTypVonEinträgen*>(*einträge*)

z.B.

```
let numbers1 = new Array<number>()
let numbers2 = new Array<number>(3)
let numbers3 = new Array<number>(345, 324, 234)
```

DOM COLLECTIONS

Nutzt man die DOM-Traversing-Methoden, um Elemente in HTML (DOM) zu selektieren, bekommt man verschiedene Strukturen zurück.

Methode `getElementsByTagName()` liefert `HTMLCollectionOf<T>` zurück.

Methode `getElementsByName()` liefert `NodeListOf<T>` zurück.

Das sind generische Strukturen. Die nicht generischen Varianten davon sind `HTMLCollection` und `NodeList`.

INTERFACES

INTERFACE > IDEA

#interface

WAS IST EINE SCHNITTSTELLE IN TS

- ein Datentyp
- eine Vorlage für später zu erstellende Klassen

WOZU IST EINE SCHNITTSTELLE

- Interfaces geben vor, aus welchen Members (Mitgliedern) besteht ein komplexer Datentyp oder eine Klasse
- Interfaces geben vor, von welchem Datentyp die einzelnen Members sind

INTERFACE AS DATATYPE

In TS ist es möglich, bei einer Variable das angelegte **Interface als Datentyp** zu benutzen.

Man braucht also **keine Klasse**, die dieses Interface implementiert.

Diese Variable ist **kompatibel mit Objekten und mit Klassen**, die (u.a.) die gleichen Member haben wie das Interface

```
interface Person {
  firstName: string;
  lastName: string;
}

function greetUser(person: Person) {
  return "Hello, " + person.firstName + " " + person.lastName;
}

let user = { firstName: "Max", lastName: "Mustermann"};

// Eine Variable, die dem Shape vom Interface 'Person' entspricht, wurde von der
Funktion akzeptiert
document.body.textContent = greetUser(user);
```

READONLY

Mit dem Modifizierwort `readonly` kontrolliert der TS-Compiler die nicht beabsichtigten Mutationen (Veränderungen) von den Eigenschaften

```
interface Todo {
  readonly text: string;
  readonly done: boolean;
}
```

In TS gibt es keine weiteren **Modifizierer bei Interfaces**, wie `public` oder `private`, die es in anderen Sprachen gibt.

DISCRIMINATED UNION TYPE

Ab TS-Version 2 (#v2.0+).

DE: Diskriminierte Vereinigung.

Andere Namen für diesen Datentyp: tagged union type (markierter Vereinigungstyp), in anderen Sprachen bekannt als 'Sum Type', 'Maybe', 'Option' oder 'Optional'.

Es handelt sich hier um eine Vereinigung von Datentypen (z.B. Interfaces), die mindestens eine gemeinsame Eigenschaft haben.

Diese gemeinsame Eigenschaft wird als 'tag' (Markierung) oder 'discriminant property' (diskriminante Eigenschaft) genannt.

Obwohl 'diskriminieren' fachlich 'unterscheiden' bedeutet, wird der Begriff 'discriminant property' in TS-Dokumentation für 'gemeinsame Eigenschaft' genutzt.

CLASSES

CLASSES > IDEA

#class

WAS IST EINE KLASSE

- eine Datenstruktur
- ein TS-Äquivalent zu den JS Objekttypen (obwohl JS auch mittlerweile Klassen hat)
- Grundlage für Objekt orientierte Programmierung

WOZU IST EINE KLASSE

- eine Klasse gibt vor, woraus die Instanzen dieser Klasse bestehen
- Konstruktor einer Klasse gibt vor, wie diese Instanzen angelegt werden
- um thematisch verbundene Daten zusammen zu halten
- um Objekte der realen Welt in der Programmiersprache abzubilden
- oft verwendet man Klassen, um Datensätze aus den Datenbanken dem Anwendungsbenutzer zu präsentieren und zur Bearbeitung bereit zu stellen

---

INSTANCE

#instance #instanz

WAS IST EINE INSTANZ EINER KLASSE

- eine Klasse in Programmierung ist nur ein Prototyp, eine Vorlage für eine Reihe von Objekten
- z.B. eine Klasse 'Person' definiert, mit welchen Informationen die konkreten Personen beschrieben werden
- diese konkreten Personen sind Instanzen der Klasse 'Person'

WAS IST EIN KONSTRUKTOR EINER KLASSE

- eine Funktion von einer besonderen Syntax
- wird genutzt, um Instanzen einer Klasse zu erzeugen (zu konstruieren)

MEMBERS

WORAUS BESTEHT EINE TS-KLASSE

- Properties / Eigenschaften
- Konstruktor(en)
- Methoden
- Accessors / Zugriffsmethoden

NICHT ERLAUBT IN EINER TS-KLASSE

- Konstanten
- globale Funktionen, wie `console.log()`

CLASS SHORTHAND

Props ohne Modifizierer sind automatisch public.

Beim Gebrauch von `public` bei den Argumenten im Konstruktor werden die entsprechenden Eigenschaften automatisch angelegt.

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

// Kurzform für:

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

Anderes Beispiel:

```
class Student {
  fullName: string;
  constructor(public firstName: string, public middleInitial: string, public
lastName: string) {
    this.fullName = firstName + " " + middleInitial + " " + lastName;
```

```
    }
  }
```

PUBLIC & PRIVATE

Private & Public Properties

```
class ClockComponent {
  private formatTime(time) {
    return …
  }
  public start() {
    …
  }
}
```

GENERICS

GENERICS > INTRO

Bei den strengtypisierten Sprachen braucht man oft eine **Funktion oder eine Datenstruktur**, die sich an **verschiedene Datentypen** beim Aufruf anpasst.

Der Entwickler möchte in diesem Fall eine Funktion / Datenstruktur erstellen, die mit einem **bestimmten Datentyp** arbeitet. **Type-checking** muss also beim Abfangen von Werten und Befüllen von Variablen aktiv bleiben.

Man weiß aber **bei der Deklaration** nicht, **welcher Datentyp** es genau sein wird.

In diesem Fall können **generische Datentypen** helfen.

Die Funktion / Datenstruktur **bindet sich** an den benötigten Datentyp erst **beim Aufruf**.

Zum Beispiel, für eine Funktion, die die Elemente von einem Array zu diesem Array nochmal hinzufügt, sollte man eine separate Definition für String-Array, eine weitere Definition für Number-Array usw. schreiben:

```
// Funktion, die an einen bestimmten Datentyp angepasst ist, hat Nachteile
let myStringArray = ['a', 'b', 'c']

let myNumberArray = [2, 3, 4]

function makeBiggerString(arr: Array<string>): Array<string>{
    let biggerArray = arr.concat(arr)
    return biggerArray
}

function makeBiggerNumber(arr: Array<number>): Array<number>{
    let biggerArray = arr.concat(arr)
    return biggerArray
}
```

GENERIC FUNCTION

Oft braucht man **Funktionen**, die **mit jedem Datentyp** arbeiten können.

Ein neuer Datentyp, wie eine Union, wird nicht helfen. Man kann nicht alles Mögliche in einer Union auflisten.

```
// Die vordefinierten Datentypen von TypeScript könnte man hier auflisten
type universal = string | number | boolean | ...
// Aber für den Fall, dass auch die vom Entwickler angelegten Datentypen
berücksichtigt werden, hilft eine Auflistung nicht mehr.
```

Eine generische Variante von der Funktion aus der vorherigen Folie ...

```
function makeBiggerString(arr: Array<string>): Array<string>{
    let biggerArray = arr.concat(arr)
    return biggerArray
}
```

... könnte so aussehen:

```
function makeBiggerGeneric<genericType>(arr: Array<genericType>):
Array<genericType>{
    let biggerArrayGeneric = arr.concat(arr)
    return biggerArrayGeneric
}
```

Die Stelle, wo man eine **Typ-Variable** definiert, ist direkt **vor den runden Klammern** für Funktionsargumente.

Der typische Name für den generischen Datentyp heißt 'T'.

```
// Identity prüft z.B. ob das Argument dem in <>-Klammern angegebenen Typ
entspricht
function identity<T>(arg: T): T { return arg; }

let output: string = identity<string>("myString");
```

Anderes Beispiel.

```
// GetProperty liefert den Wert von der gewünschten Eigenschaft bei einem Objekt:
function getProperty<T, K extends keyof T>(obj: T, key: K) { return obj[key]; }
```

DECORATORS

> DECORATORS > IDEA

#decorator

WAS IST EIN DEKORATOR

- **eine Funktion**, die an Klassen bzw. deren Members oder auch an den Funktionen-Parametern angebunden wird

WOZU IST EIN DEKORATOR

- Mit Dekoratoren lassen sich **Funktionen und Klassen** nach ihrer Instanziierung oder Aufruf **verändern**
- Mit Dekoratoren können Props, Methoden oder Parameter mit **Metainformationen** markiert werden
- Mit Dekoratoren können **Daten**, die an eine Klasse als Prop-Values oder an eine Funktion als Parameter übergeben werden, **abgefangen werden**
- Anwendungsbeispiele: **Logging**, **Caching**, **Data Validation**

Wenn man eine **JS-Bibliothek** (oder Framework) schreibt, dann kann man mit Hilfe von **Dekoratoren zusätzliche Werkzeuge** anlegen, die schnell eingesetzt werden können.

Um Dekoratoren zu aktivieren, muss eine Änderung an **tsconfig** vorgenommen werden:

```
tsc --target "ES5" --experimentalDecorators
```

Semantische Syntax für **Aufruf** eines Dekorators z.B. bei Klassen

```
@expression // expression muss eine Funktion sein oder eine Funktion liefern
class MyClass {
  // ...
}
```

Definiert werden die Dekoratoren wie ganz normale Funktionen:

```
function doSmthWithClass(targetClass) {
  // changes for targetClass
}
```

Als **Argument** an diese Dekorator-Funktion wird die **Klasse** übergeben, an der dieser Dekorator angwendet wird:

```
// Definition vom Dekorator
function setIdTo100(target: Function) { // Klassen sind im kompilierten Code Funktionen
  target.prototype.id = 100
}

// Aufruf vom Dekorator
@setIdTo100
class TestClass {
  id: number;
}

console.log(new TestClass().id) // druckt 100
```

---

4 Arten von Dekoratoren:

- Klassen-Dekoratoren
- Eigenschaften-Dekoratoren (prop decorators)
- Methoden-Dekoratoren
- Parameter-Dekoratoren

Es können **mehrere Dekoratoren** an einem Objekt angewendet werden. Sie werden dann der Reihe nach ausgeführt.

DECORATOR FACTORY

Wenn ein Dekorator Parameter erwartet, nutzt man 'Dekoratoren-Fabrik'.

In dem Fall ist es syntaktisch eine Funktion die eine andere Funktion zurückgibt:

```
function setId(options: {id: number}) {
  return function (target: Function) {
    target.id = options.id;
  }
}

@setId({id: 100})
class TestClass {
  id: number;
```

```
}

console.log(new TestClass().id) // druckt 100
```

DECORATORS > EXAMPLE

In Angular werden Dekoratoren verwendet, um Metadaten bei einer Klasse zu ergänzen:

```
@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
})
export class AppComponent {
    name = 'Max';
    // ...
}
```

DEMO

siehe Demo-Datei zu Themen

- Methodendekorator
- Prop-Dekorator
- Param-Dekorator

MORE TYPES

UNKNOWN

- unknown = [*primitive* | object]
- [any | unknown] = unknown

NEVER

- never != [*primitive* | object] (nothing is assignable to never)
- [*primitive* | object] = never (never is assignable to everything)

ENUMS

'enum' ist Abkürzung für 'enumeration' (Aufzählung).

'Enumerated type' auf Deutsch: Aufzählungstyp.

Es lässt sich wie eine Sammlung von genannten Werten beschreiben.

```
// Ziel: einheitlichkeit bei bestimmten Werten innerhalb von einem Team
enum lengthUnit {cm, meter, kilometer, mile}

let shiffGeschwindigkeitEinheit: lengthUnit = lengthUnit.mile
```

EXTENDING TYPES

```
type AsyncReturnType<T extends (...args: any) => any>
```

INFERING TYPES

```
T extends (...args: any) => infer U ? U :
any
```

https://www.jpwilliams.dev/how-to-unpack-the-return-type-of-a-promise-in-typescript

---

KEYOF

Mit dem Operator 'keyof' bekommt man ein **Union-Type**, bestehend **aus Keys** von einem Objekttyp.

```
type Person = {firstname: string; lastname: string};
type Personkeys = keyof Person; // eine Union 'firstname' | 'lastname'
```

DECLARATIONS

http://definitelytyped.org/

FACTS

TS VERSIONS

Datum	Version
2012, 1. Okt	erschienen
2014, 6. Okt	1.1
2016, 22. Sep	2.0
2018, 30. Jul	3.0
2020, 21. Aug	4.0

TS COMMUNITY

TWITTER https://twitter.com/typescript/

INDEX

HASHTAGS

Um Definition von folgenden Begriffen schnell in den Folien zu finden, geben Sie in die Suchfunktion (`strg` + `F`) ein: # und etwas aus der Liste ein:

- class
- decorator
- DOM-quering
- instance
- instanz
- interface
- seitV3.5
- seitV3.6
- ts
- tsc
- tools
- traversing