

TypeScript



Markus Völk - 25.10.2023

Markus Völk

- Seit 2012 bei ZwickRoell
- Abteilung: Zentralentwicklung
- Team-Product
- DHBW Heidenheim Dozent
- Themen
 - ▶ TypeScript
 - ▶ nest.js
 - ▶ Angular
 - ▶ Nx
 - ▶ SoftwareEngineering



Agenda

- Vorstellung Teilnehmer
- tsconfig.json einrichten
- Basic Types
- Klassen
- Objekt Typen
- Control Flow
- Typen aus Typen erzeugen
- ... Wünsche?

Einstieg

Einstieg JavaScript



JS

- Existiert seit 1995
- Ursprünglich als clientseitige Schriftsprache entwickelt
- Alle Browser unterstützen seit langem JavaScript
- Seit 2009 wird Node.js entwickelt, welches JavaScript serverseitig nutzbar macht.
- Die aktuellste Spezifikation ist ECMAScript 2023
- Unterstützt dynamische Typisierung

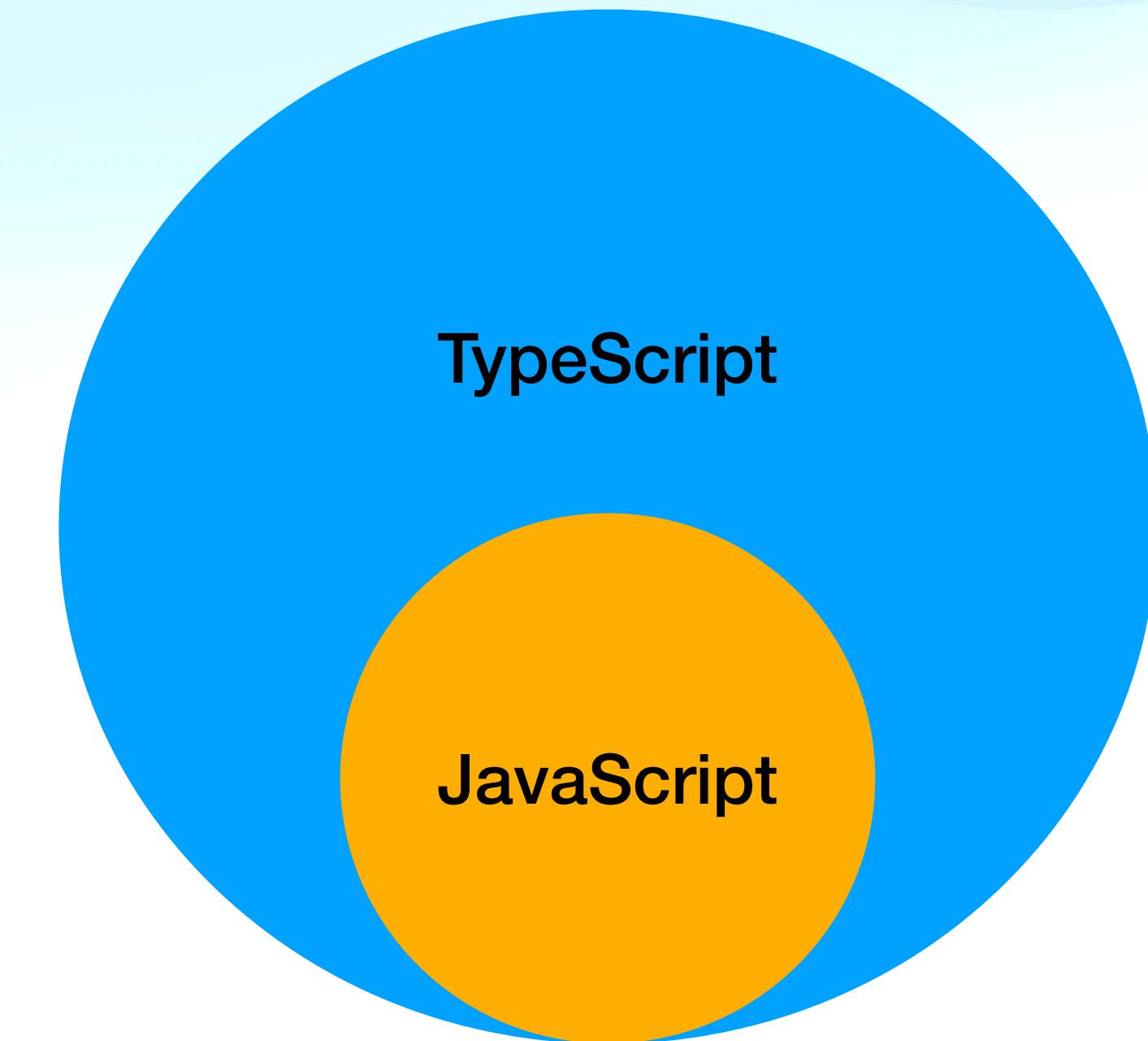
Einstieg TypeScript



- „TypeScript is JavaScript with syntax for types“
- Wird seit 2012 entwickelt
- Aktuell ist Version 5.2 (ECMAScript 2022 wird voll unterstützt)
- Open-Source (<https://github.com/microsoft/TypeScript>)
- TypeScript wurde von Anders Hejlsberg entwickelt (C#)
- Heute wird die TypeScript Entwicklung von Microsoft geführt

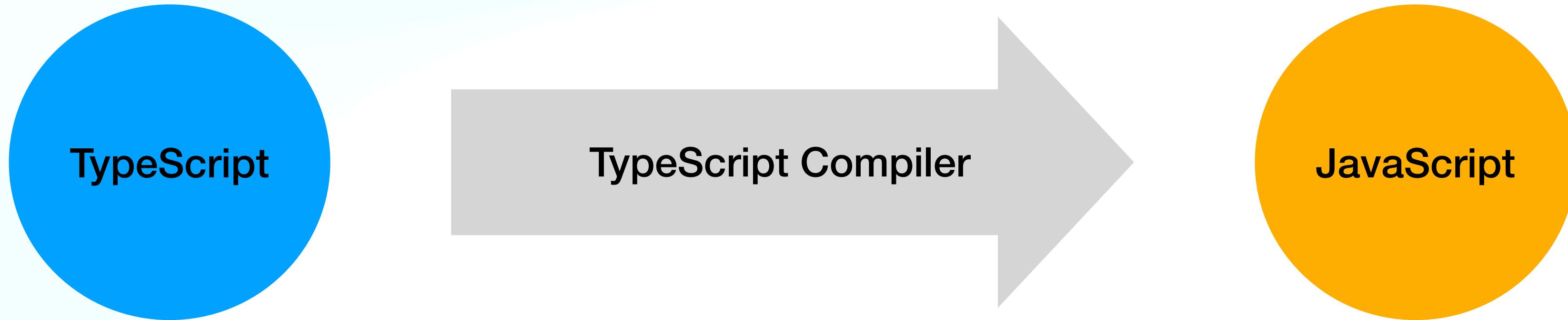
Einstieg

- Jedes JavaScript Programm ist auch ein TypeScript Programm



Einstieg

- Der TypeScript Compiler entfernt alle Typen und generiert somit JavaScript (vereinfacht)



Einstieg JavaScript-Code

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

3
12
12
12

```
console.log(add(1, 2));  
console.log(add("1", "2"));  
add("1", 2);  
add("1", "2");
```

Einstieg TypeScript-Code



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



```
console.log(add(1, 2));  
console.log(add("1", "2"));  
console.log(add(1, "2"));  
console.log(add("1", 2));
```



```
error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.  
6 console.log(add("1", "2"));
```

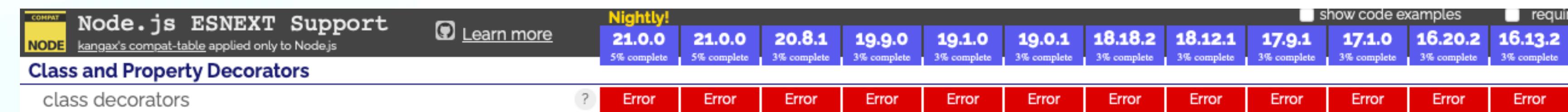
Einstieg

Vorteile TypeScript gegenüber JavaScript

- Statische Typisierung
- Neue Features sind nutzbar
- Deutlich bessere IDE Unterstützung (Refactoring, Verwendungsstellen, ...)
- Weniger Fehleranfällig
- Sicherer Refactoring
- Ziel-JavaScript Version (fast) unabhängig von der Implementierung

Einstieg Neue Features

- TypeScript unterstützt neue JavaScript Features noch bevor andere Umgebungen diese implementieren (ab Stage 3)
 - Beispiel:
 - (Function-, Class-, ...) Decorators werden von TypeScript unterstützt (Stage 2 experimental ab TypeScript 5.0 Stage 3)



Einstieg

Vorteile in Business-Anwendungen

- Erfahrungsberichte zeigen, dass etwa 15-40% der Bugs mit TypeScript vermieden werden können
- Zeitersparnis bei der Entwicklung
 - Weniger Bugs, weniger Zeit für QA und Testing
 - Code ist lesbarer und von Entwicklern einfacherer zu verstehen
- Einfache Migration existierender JavaScript Projekte
- TypeScript als Full-Stack Programmiersprache
- Abwärtskompatibilität: (meistens) kann TypeScript in ES3 transpiliert werden
- Robuster, weniger Fehleranfälliger Code

Einstieg

Vorteile in Business-Anwendungen

- Großes Ökosystem von JavaScript / npm kann genutzt werden
- Projekte / Unternehmen, welche TypeScript nutzen:

- Slack
- Angular
- Ionic
- VSCode
- Airbnb
- Google



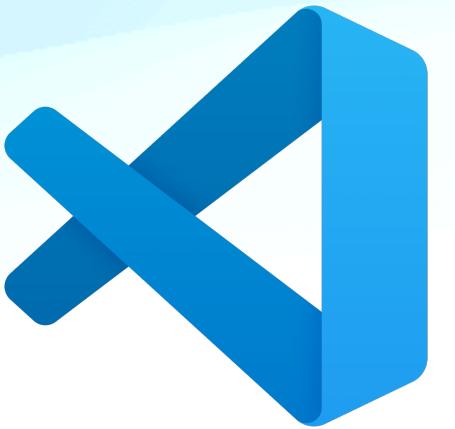
- Vue
- Deno
- GitHub
- Microsoft
- React
- Postman



Setup

Benötigte Tools

- Node.js / nvs / nvm > aktuelle LTS ist v20
- TypeScript compiler > *npm install typescript*
- IDE: Visual Studio Code / WebStorm



Beispiele

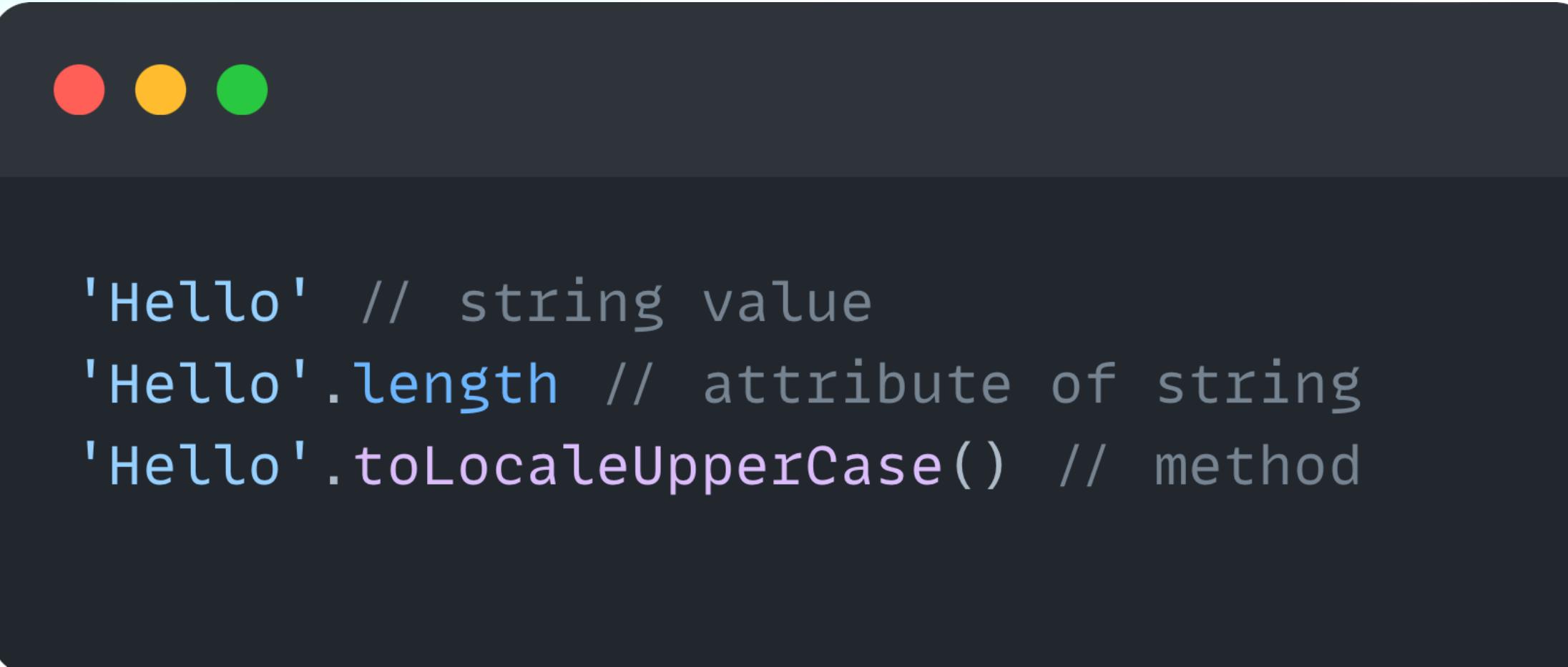
Übung

- JavaScript dynamische Typisierung
- Hello World
- Migration eines JavaScript Projekts

TypeScript Basics

Typen

- Ein Typ ist ein Weg, *Attribute* und *Funktionen* einer *Variablen* zu beschreiben
- Jede Variable hat einen Typ. Implizit oder explizit



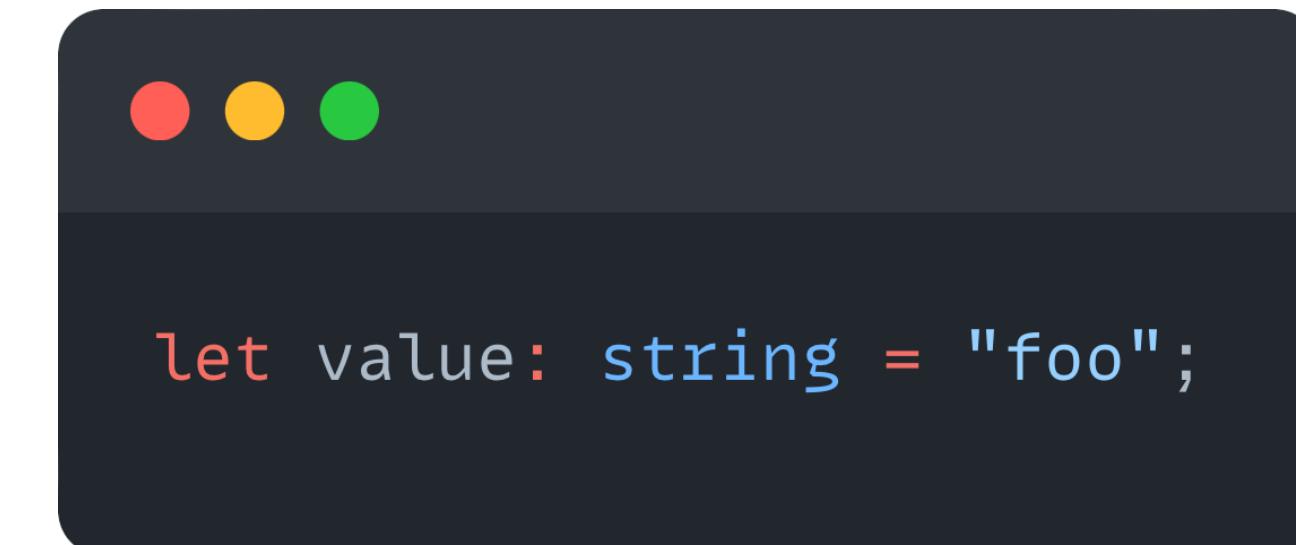
A screenshot of a dark-themed code editor window. At the top left, there are three small colored circles: red, yellow, and green. The main area contains the following TypeScript code:

```
'Hello' // string value
'Hello'.length // attribute of string
'Hello'.toLocaleUpperCase() // method
```

TypeScript Basics

Typen

- Primitive Typen
 - string
 - number
 - boolean
 - null
 - undefined
 - symbol
 - bigint (ES2020)
 - literal
- Objekt Typen
 - function
 - array
 - class
 - object
 - interfaces



TypeScript Basics

Arrays

- Syntax: „type[]“ oder „Array<type>“
- Achtung: Nicht mit Tuples („[type]“) verwechseln!



```
let names: string[];
names = ["Manfred", "Michael"];
names.push("Manuela");
names.length; // 3
```

TypeScript Basics

Objekte

- Syntax: „{ key: type }“
- Optionale keys mit „?“
- Empty-Object Typ: „{}“



```
let person: { name: string };
person = { name: "Manfred" };
person.name;
```

TypeScript Basics

Funktionen

- Syntax: „(arg1: type) => type“
- Optionale Argumente mit „?“
- „Contextual typing“

```
let convert: (name: string) => string;
convert = function(name: string) {
    return name.toUpperCase();
}
convert = function(name: string) {
    console.log(name);
}; // < Error!
convert(123); // < Error!
```

```
const names = ["Manfred", "Manuela"];
names.forEach(function (s) { // type for s is inferred from context
    console.log("Hi", s);
});
```

TypeScript Basics

Any

- Ein spezieller Typ ist „any“
- Any deaktiviert alle Type Checks für die Variable
- Compiler Flag „noImplicitAny“
- Weitere spezielle Typen:
 - „unknown“ und „never“

```
let obj: any = { x: 0 };
// None of the following lines of code will throw compiler errors.
// Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

TypeScript Basics

Literals

- Zusätzlich zu den generischen Typen „string“ und „number“ gibt es noch literals, welche genau einen spezifischen Wert annehmen können.

```
const str1 = "hello"; // ← str1 is type "hello"
let str2 = "hello"; // ← str2 is type string
let str3: "hello" = "hello"; // ← str3 is type "hello"

const num1 = 1; // ← num1 is type 1
let num2 = 1; // ← num2 is type number
let num3: 1 = 1; // ← num3 is type 1

function print(str: string, alignment: "left" | "right" | "center") {
    // ...
}

let alignment = "left";
print("a", alignment); // < Argument of type 'string' is not assignable
// to parameter of type '"left" | "right" | "center"' .ts(2345)
```

TypeScript Basics

Interfaces

- Ein Interface kann auch genutzt werden, um Objekt Typen zu definieren
- Interfaces sind erweiterbar („Declaration Merging“)

```
interface Person {  
    name: string;  
}  
interface Person {  
    age: number;  
}
```

```
interface Person {  
    name: string;  
    age: number;  
}  
  
function greet(person: Person) {  
    // ...  
}
```

TypeScript Basics

Typ Alias

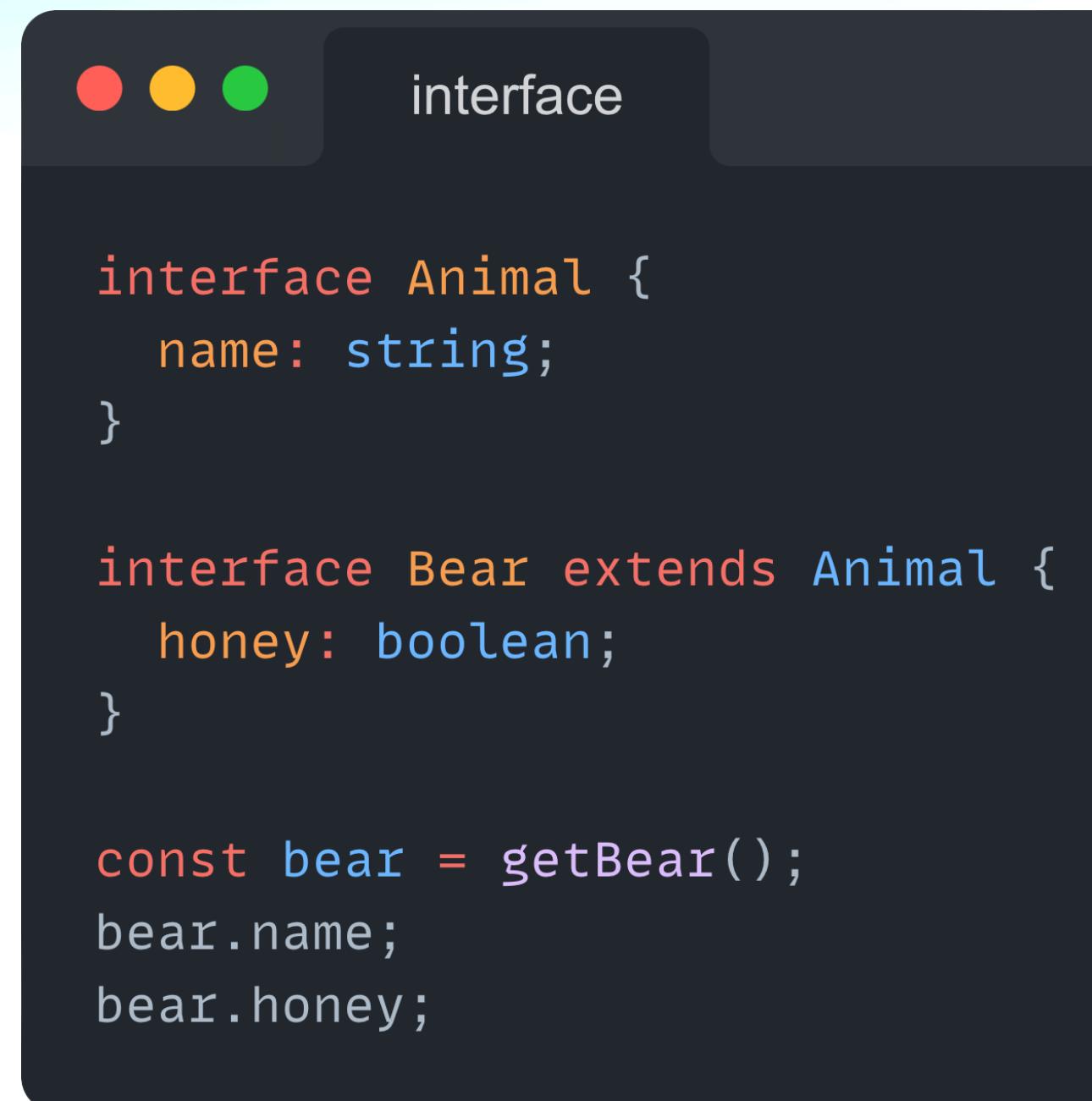
- Wenn ein spezieller Typ mehrmals genutzt werden soll, so kann daraus ein Typ-Alias erzeugt werden
- Der Typ Alias ist wirklich nur ein Alias, als würde man den eigentlichen Typen anstelle des Alias nutzen

```
type Point = {  
    x: number;  
    y: number;  
};  
type ID = number | string;
```

TypeScript Basics

Vergleich interface <> Typ Alias

- Ein Typ Alias für ein Objekt Typ und Interfaces sind sehr ähnlich
- Unterschied: Ein Typ Alias kann nicht erweitert werden
- Empfehlung von typescriptlang.org:
„If you would like a heuristic, use interface until you need to use features from type“



```
● ● ● interface
interface Animal {
  name: string;
}

interface Bear extends Animal {
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```



```
● ● ● type alias
type Animal = {
  name: string;
}

type Bear = Animal & {
  honey: boolean;
}

const bear = getBear();
bear.name;
bear.honey;
```

TypeScript Basics

Type Inference

- Explizite Typ Annotationen sind nicht (immer) notwendig.
- Es wird der am besten zutreffende Typ automatisch gewählt
- Wenn eine Variable mit einem Wert initialisiert wird, wird dessen Typ für die Variable übernommen
- Dies gilt auch für Funktionsargumente mit default-Werten



```
let x = [0, 1, null];
// (number | null)[]
```



```
function addValue(value=100) {
    // value is of type number
}
```



```
let name: string;
name = "Manfred";

let age = 3;
age = "2"; // < Error

let city; // implicit any
city = "Ulm";
city = 89079;
```

TypeScript Basics

Type Widening

- Wenn einer Variable ein Literal Type zugewiesen wird, nutzt TypeScript den allgemeineren Typen
- Wenn einer Konstanten ein Literal Type zugewiesen wird, nutzt TypeScript den Literal Type

```
let direction = "north";
// type = string // type widening
const direction2 = "north";
// type = "north"
```

TypeScript Basics

Type Narrowing

- Gegenteil von Type Widening
- Basierend auf Conditions (z.B. if Statements) wird der Typ einer Union eingeschränkt

```
function processDirection(direction: "north" | "south" | "east" | "west") {  
    // type direction = "north" | "south" | "east" | "west"  
    if (direction === "north" || direction === "south") {  
        // type direction = "north" | "south"  
    } else {  
        // type direction = "east" | "west"  
    }  
}
```

TypeScript Basics

Union types

- Union kombiniert Typen
- Begriff kommt aus der Typentheorie
- Resultat ist die **Schnittmenge** aller Typen der Union

```
function getUser(id: string | number) {  
    // ...  
}
```

TypeScript Basics

Intersection types

- Intersection kombiniert Typen
- Begriff kommt aus der Typentheorie
- Resultat ist die **Vereinigungsmenge** aller Typen der Intersection
- Mit einer intersection können Typen erweitert werden



```
function notify(user: User & Loggable) {  
    // ...  
}
```

TypeScript Basics

Enum

- Numerische Enums
- String Enums
- Mixed Enums
- Enums können berechnete Werte haben
- Zur Laufzeit werden aus Enums Objekte
- const enums
- In Modernes TypeScript ist ein Object „as const“ oft auch ausreichend

TypeScript Basics

Übung

- Schreiben Sie eine Funktion, welche eine Zahl entgegennimmt und „Even“ für gerade Zahlen und „Odd“ für ungerade Zahlen zurückgibt.
- Schreiben Sie ein Programm, welches eine JSON Datei einliest und die Attribute des JSON Objekts rekursiv auf der Konsole ausgibt.

Klassen

- „class“ keyword aus ES2015 wird voll unterstützt
- TypeScript erweitert JavaScript Klassen
- Weitere Beispiele im Live-Code Beispiel

Objekt Typen

- Objekt Typen können auch als anonyme Typen verwendet werden (ohne type alias oder interface)
- Modifiers: Optional, Readonly
- Objekt Typen können eine „Index Signature“ haben
- Type Checks auf Objekt-Typen hängen davon ab, wo das Objekt erzeugt und wie es verwendet wird

Objekt Typen

Array-Type

- Array Types:
 - `string[]` ist eine Kurzform für `Array<string>`
- `ReadonlyArray`
 - Entspricht einem Array-Typen, der nicht verändert werden kann

Objekt Typen

Tuples

- Tuples sind eine Sonderform von Arrays
- Sie beschreiben die Länge und der Typ der Werte an fixierten Stellen im Array

```
type Pair = [string, number];
const p: Pair = ["Manfred", 25];
type First = typeof p[0]; // type string
type Second = typeof p[1]; // type number
type Third = typeof p[3]; // Tuple type 'Pair' of length '2' has no element at index '3'

type PairWithBooleans = [string, number, ...boolean[]];
```

Control Flow

Type Narrowing

- Es gibt verschiedene Möglichkeiten, einen Typen im Programmfluss einzuschränken („type guards“)
 - „typeof“ Check
 - if Statements
 - „type predicates“ Returnwerte
 - assertion Funktionen
 - Discriminated unions
- Durch type narrowing kann ein Typ auch zu „never“ reduziert werden

Typen aus Typen erzeugen

Generics

- Generics sind Typen, welche Parameter übergeben bekommen
- Beispiel identity function



```
function identity(arg: number): number {
    return arg;
}

function identity(arg: any): any {
    return arg;
}

// ...
// as generic
function identity<Type>(arg: Type): Type {
    return arg;
}
```

Typen aus Typen erzeugen

Keyof type Operator

- „keyof“ repräsentiert die Union aller keys eines Objekt Typen
- Bei Objekt Typen mit Index Signatur liefert keyof entsprechend „string“ oder „number“



```
type Point = { x: number; y: number };
type P = keyof Point; // "x" | "y"

type ArrayType = { [key: number]: unknown };
type A = keyof ArrayType; // number

type MapType = { [key: string]: unknown };
type M = keyof MapType; // string | number
```

Typen aus Typen erzeugen

Typeof type Operator

- „typeof“ von JavaScript kann nur zur Laufzeit genutzt werden
- TypeScript ergänzt den typeof Operator als Typ

```
const foo = "bar";
type t1 = typeof foo; // string

function baz() { return 123; }
type Result = ReturnType<typeof baz>;
// ReturnType<baz> erzeugt einen Fehler
```

Typen aus Typen erzeugen

Index Access Typ

- Über einen Index Access Type können wir auf Typen von Attributen eines Objekt Typs zugreifen

```
interface Person {  
    name: string;  
    age: number;  
    isFemale: boolean;  
}  
  
type Type1 = Person["age" | "name"]; // number | string  
type Type2 = Person[keyof Person]; // number | string | boolean  
type SomeKeys = "a" | "name";  
// type Type3 = Person[SomeKeys]; // Property 'a' does not exist on type 'Person'  
type Type4 = Person[SomeKeys & keyof Person]; // string  
type Type5 = Person[Exclude<SomeKeys, "a">]; // string
```

Typen aus Typen erzeugen

Conditional Types

- Mit conditional Types können typen Abhängig von einem Eingangstyp bestimmt werden
- Findet vor allem bei Generics Anwendung

```
type DocumentOrFile<T> = T extends string ? Document : File;  
  
// conditional type constraints  
type IdOf1<T extends { id: unknown }> = T["id"];  
type IdOf2<T> = T extends { id: unknown } ? T["id"] : never;  
  
// infer  
type Flatten2<T> = T extends Array<infer ItemType> ? ItemType : T;  
  
// conditional type distribution  
type ToArray<Type> = Type extends unknown ? Type[] : never;  
type StrArrOrNumArr = ToArray<string | number>; // type string[] | number[]
```

Typen aus Typen erzeugen

Mapped Types

- Um aus Objekt Typen neue Objekt Typen zu erzeugen, können Mapped Types genutzt werden

```
type Person = {
    name: string;
    age: number;
    isFemale: boolean;
};

type Subscription<T extends object> = {
    [K in keyof T]: () => T[K]; // map to function which returns the type of the attribute
};

type AsGetters<Type> = {
    [Property in keyof Type as `get${Capitalize<string & Property>}`]: () => Type[Property];
};

type WithoutName<T> = {
    [K in keyof T as Exclude<K, "name">]: T[K];
};
```

Utility Typen

Awaited<Type>

- Entpackt den Typ aus einer Promise

```
type ReturnType = Promise<string>;
type S = Awaited<ReturnType>;
// S = string
```

Utility Typen

Partial<Type>

- Setzt alle Attribute eines Objekt Typen auf Optional

```
type Person = {  
    name: string;  
    age: number;  
}  
type PersonInput = Partial<Person>;  
// {  
//     name?: string;  
//     age?: number;  
// }
```

Utility Typen

Required<Type>

- Setzt alle Attribute eines Objekt Typen auf „required“

```
type PersonInput = {
  name?: string;
  age?: number;
}

type Person = Required<PersonInput>;
// {
//   name: string;
//   age: number;
// }
```

Utility Typen

Readonly<Type>

- Setzt alle Attribute eines Objekt Typen auf „readonly“

```
type Person = {  
    name: string;  
    age: number;  
}  
  
type PersonFixed = Readonly<Person>;  
// {  
//     readonly name: string;  
//     readonly age: number;  
// }
```

Utility Typen

Record<Keys, Type>

- Erzeugt einen Objekt Typen mit den angegebenen keys, deren Werte Type entsprechen

```
type SchemaEntry = { type: "string" | "number" };
type Person = {
    name: string;
    age: number;
}
type PersonSchema = Record<keyof Person, SchemaEntry>;
// type PersonSchema = {
//     name: { type: "string" | "number" };
//     age: { type: "string" | "number" };
// }
```

Utility Typen

Pick<Type, Keys>

- Erzeugt einen neuen Objekt Typen, der **nur** die angegebenen keys enthält

```
type Person = {  
    name: string;  
    age: number;  
    isFemale: boolean;  
}  
  
type MinPerson = Pick<Person, "name" | "age">;  
// type MinPerson = {  
//     name: string;  
//     age: number;  
// }
```

Utility Typen

Omit<Type, Keys>

- Erzeugt einen neuen Objekt Typen, der die angegebenen keys **nicht** enthält

```
type Person = {  
    name: string;  
    age: number;  
    isFemale: boolean;  
}  
  
type MinPerson = Omit<Person, "isFemale">;  
// type MinPerson = {  
//     name: string;  
//     age: number;  
// }
```

Utility Typen

Exclude<UnionType, ExcludedMembers>

- Generiert einen neuen Union Type, welcher die ExcludedMembers nicht enthält

```
type Person = {  
    name: string;  
    age: number;  
    isFemale: boolean;  
}  
type SomeAttributes = Exclude<keyof Person, "name">;  
// "age" | "isFemale"
```

Utility Typen

Exclude<UnionType, ExcludedMembers>

- Generiert einen neuen Union Type, welcher die ExcludedMembers nicht enthält

```
type Manfred = {
    name: "Manfred";
    age: number;
    isFemale: boolean;
};

type KeysWithoutName = Exclude<keyof Manfred, "name">;
// "age" | "isFemale"

type Manuela = {
    name: "Manuela";
    age: number;
    isFemale: boolean;
};

type PersonsWithoutManni = Exclude<Manfred | Manuela, { name: "Manfred" }>;
```

Utility Typen

Extract<Type, Union>

- Generiert einen neuen Union Type, alle Elemente enthält, die in Union enthalten sind

```
type Manfred = {
    name: "Manfred";
    age: number;
    isFemale: boolean;
};

type KeysWithoutName = Extract<keyof Manfred, "name">;
// "name"

type Manuela = {
    name: "Manuela";
    age: number;
    isFemale: boolean;
};

type OnlyManni = Extract<Manfred | Manuela, { name: "Manfred" }>;
```

Utility Typen

NonNullable<Type>

- Entfernt null und undefined aus „Type“

```
type R = NonNullable<string | number | undefined | () => void;  
// string | number | () => void
```

Utility Typen

Parameters<Type>

- Erzeugt einen Tuple Typen aus allen Parameter eines Funktionstypen

```
type T0 = Parameters<() => string>;
// T0 = []
type T1 = Parameters<(s: string) => void>;
// T1 = [s: string]
```

Utility Typen

ConstructorParameters<Type>

- Erzeugt einen Tuple Typen aus allen Parameter eines Konstrukts

```
type T0 = ConstructorParameters<ErrorConstructor>;
// T0 = [message?: string | undefined]

class C {
    constructor(a: number, b: string) {}
}

type T3 = ConstructorParameters<typeof C>;
// T3 = [a: number, b: string]
```

Utility Typen

ReturnType<Type>

- Extrahiert den ReturnType aus einem Funktionstypen

```
type T0 = ReturnType<() => string>;  
// T0 = string  
type T1 = ReturnType<<T extends number[]>() => T>;  
// T1 = number[]
```

Utility Typen

InstanceType<Type>

- Erzeugt einen Typen, der dem Instanz-Typen eines Konstrukteurs entspricht

```
class C {}

type Ctor = typeof C;
type T0 = InstanceType<Ctor>
// T0 = C
```

Utility Typen

ThisParameterType<Type>

- Extrahiert den Typen des this Parameters eines Funktionstypen
- Gibt unknown zurück, wenn der Funktionstyp kein this Typ hat.

```
function toHex(this: Number) {  
    return this.toString(16);  
}  
  
function numberToString(n: ThisParameterType<typeof toHex>) {  
    return toHex.apply(n);  
}
```

Utility Typen

OmitThisType<Type>

- Erzeugt einen Funktions-Typ ohne this Parameter

```
function toHex(this: Number) {  
    return this.toString(16);  
}  
  
const fiveToHex: OmitThisParameter<typeof toHex> = toHex.bind(5);  
  
console.log(fiveToHex());
```

Utility Typen

ThisType<Type>

- Nur verwendbar, wenn „noImplicitThis“ aktiviert ist
- Verwendet in einer Intersection mit einem Funktionstypen, markiert ThisType den neuen Funktionstypen mit dem This Typen des Funktionstypen, der übergeben wird

Modules

- In TypeScript können ESMModules verwendet werden. Aus solch einem Modul kann auch ein Typ erzeugt werden
- Typ beinhaltet: { ...named, default }



```
type HelloWorldModule = typeof import("./module.js");
```

Weiterführende Links

- TypeScript Handbuch: <https://www.typescriptlang.org/docs/handbook/intro.html>
- Beispiele Exclude<Type>: <https://www.totalltypescript.com/uses-for-exclude-type-helper>