Allgemeines

- Syntaktische Obermenge von JavaScript
- Fügt statische Typisierung hinzu
- Ermöglicht Angabe der Datentypen
- Typüberprüfung erfolgt vorab während Kompilierung
- JavaScript:
  - Sprache mit loser Typisierung
  - Typ von Variablen und Funktionsparametern wird durch ihre Verwendung festgelegt
- TypeScript Compiler wandet TypeScript Code in JavaScript Code um
- Installation des Compilers mittels node.js Package Manager (npm)
- Ausführung des Compilers mittels npx
- TypeScript-Compiler wird durch Datei tsconfig.json konfiguriert
- Initiierung durch das Kommando npx tsc -init
- Festlegung z.B. von Eingabe- und Ausgabeverzeichnissen

```
{
    "include": ["src"],
    "compilerOptions": {
        "outDir": "./build"
    }
}
```

Typen

- Typen in TypeScript können explizit oder implizit festgelegt werden
- Explizite Festlegung: Variablenname gefolgt von Doppelpunkt und Typ
- Typen: **number, string, boolean**
- Bei impliziter Typ-Festlegung schließt TypeScript den Typ auf Basis der ersten Zuweisung
- In beiden Fällen ist der Typ der Variablen hinterher String (Zeichenkette)

```
// explizite Typ-Festlegung
let vorname: string = "Olaf";

// implizite Typ-Festlegung
let vorname2 = "Friedrich";
```

- Spezielle Typen: **any** und **unknown**
- Wenn TypeScript auf einen Typ nicht schließen kann, wird die Typisierung hierfür außer Kraft gesetzt (Typ: **any**)
- Verhindert keine falsche Nutzung
- Besser: Typ explizit auf unbekannt (**unknown**) setzen und vor der Verwendung ggf. auf korrekten Typ casten

```
let ergebnis: any = 33;
ergebnis = "Christian"; // kein Fehler!
if (ergebnis == false)  // kein Fehler!
    ergebnis = true;

let resultat: unknown = 11;
```

- Fehler bei Typ-Wechsel: Fehlermeldung, wenn Typ eines zugewiesenen Wertes und zuvor (explizit oder implizit) festgelegter Typ nicht übereinstimmen!

```
// explizite Typ-Festlegung
let vorname: string = "Olaf";
vorname = 3.14;

// implizite Typ-Festlegung
let vorname2 = "Friedrich";
vorname2 = true;
```

Arrays

- Zusätzliche eckige Klammern nach dem Typ
- Alle Array-Elemente haben den gleichen Typ!
- Auch bei Arrays kann die Typ-Zuweisung implizit erfolgen
- Arrays, deren Inhalte nicht verändert werden können, verwenden zusätzlich das Attribut **readonly**

```
const namen: string[] = [];
// JS: const name = [];
name[0] = "Robert";
name[1] = 42;

const vornamen: readonly string[] = [];
vornamen[0] = "Annalena";
```

Tupel

- Typisiertes Array mit einer vordefinierten Länge und festgelegten Typen
- Für jeden Index wird der Typ vorab explizit festgelegt
- Für höhere Indexwerte ist der Typ immer **any**
- Höhere Indexwerte ausschließen mit **readonly**

```
let meinTupel: [string, number, boolean];
meinTupel = ['Hubertus', 51, true];
meinTupel[2] = 12; // Fehler!
meinTupel[4] = "Boris"; // kein Fehler!

let meinAnderesTupel: readonly [string, number];
meinAnderesTupel = ['Nancy', 53];
meinAnderesTupel[2] = true; // Fehler!
```

Objekte

- Typ für jede Eigenschaft wird individuell festgelegt
- Zuweisungen falschen Typs führen zu Fehlermeldungen

```
const kaffeeautomat: {marke: string, modell: string,
seriennr: number} = {
    marke: "Jura",
    modell: 'F9',
    seriennr: 3318482
};
```

- Alle definierten Felder müssen initialisiert werden!
- Optionale Felder können entsprechend mit **?** definiert werden

```
const kaffeeautomat: {marke: string, modell: string,
seriennr: number} = {
    marke: "Jura",
    modell: 'F9'
};

const kaffeeautomat: {marke: string, modell: string,
seriennr?: number} = {
    marke: "Jura",
    modell: 'F9'
};
```

Aufzählungen

- Repräsentieren Gruppen von Konstanten
- Aufzählungen können numerisch sein oder auf Zeichenketten beruhen
- Standardmäßig hat die erste Konstante den Wert 0, die zweite 1, die dritte 2, etc.
- Es ist möglich, einzelne Werte anzupassen, nachfolgende Werte sind jeweils 1 höher
- Alle Konstanten müssen jedoch unterschiedliche Werte haben

```
enum richtung {
    VORWAERTS, RUECKWAERTS, LINKS, RECHTS, HOCH, RUNTER
};
// VORWAERTS = 0 ... RUNTER = 5

enum farbe { ROT = 1, GRUEN, GELB, BLAU, ORANGE = 10
};
//GRUEN = 2 ... BLAU = 4
```

- Alternativ können Aufzählungen Zeichenketten enthalten
- Zugriff auf einzelne Konstanten mittels Punktoperator

```
enum wochentag {
    MONTAG = 'Montag',
    DIENSTAG = 'Dienstag',
    MITTWOCH = 'Mittwoch',
    DONNERSTAG = 'Donnerstag',
    FREITAG = 'Freitag',
    SAMSTAG = 'Samstag',
    SONNTAG = 'Sonntag'
};

foo(wochentag.FREITAG); // Parameter "Freitag"
```

Typen Aliase

- Für die weitere Verwendung kann man Typen benennen und anschließend wie Basistypen verwenden
- Insbesondere sinnvoll für komplexere Typen (Arrays, Tupel, Objekte, Aufzählungen)

```
type Person = [string, string, number];
type Alter = number;

let km: Person = ["Mustermann", "Klaus", 40];
let alterKM: Alter = 40;
```

Interfaces

- Ähnlich zu Typen-Aliasen, aber ausschließlich für Objekte

```
interface Espressomaschine {
    marke: string,
    modell: string,
    serienNr: number
};

const meinKA: Espressomaschine {
    marke: "De'Longhi",
    modell: "La Specialista Maestro",
    serienNr: 39472833
};
```

- Können um weitere Eigenschaften erweitert werden

```
interface Siebtraeger extends Espressomaschine {
    filterDurchmesser: number,
    kaffeeMuehle: boolean
};

const meinKA: Siebtraeger {
    marke: "De'Longhi",
    modell: "La Specialista Maestro",
    serienNr: 39472833,
    filterDurchmesser: 51
};
```

Vereinigungen

- Können Variablen mehr als einen Typ haben, spricht man von Vereinigungen (unions)
- Zulässige Typen werden mit einem Oder-Operator — voneinander getrennt

```
let switch: number | boolean = 1;
switch = false; // korrekt!
switch = 'an'; // Fehler!
```

Funktionen

- Der Typ der einzelnen Parameter und der Typ des Rückgabeparameters können festgelegt werden
- Werden keine Typen festgelegt, wird **any** angenommen
- Eine Rückgabetyp **void** bedeutet, dass die Funktion keinen Wert zurückgibt

```
function Hallo(): void {
    alert('Hallo!');
}

function Max(wert1: number, wert2: number): number {
    return wert1 > wert2 ? wert1 : wert2;
}
```

- Optionale Parameter werden mit einem ? hinter dem Namen gekennzeichnet
- Optionale Parameter können nur am Ende stehen
- Default-Parameter sind ebenfalls optional, haben in diesem Fall jedoch einen fest vorgegebenen Wert
- Dieser wird über eine Zuweisung im Funktionskopf festgelegt

```
function Summe(wert1: number, wert2: number, wert3?: number)
: number {
    return wert1 + wert2 + (0 || wert3);
}

function Wurzel(a: number, n: number = 2): number {
    return a ** (1/n);
}
```

Casting

- Typ einer Variable explizit umdeuten
- Variable wird **as** gefolgt vom gewünschten Typ nachgestellt
- Funktioniert nur, wenn Inhalt der Variablen dies auch erlaubt

```
let a: unknown = 'Hallo!';
alert(a as string);
```

Klassen in TypeScript

- Eigenschaften haben wie bei Objekten Typen
- Eigenschaften und Funktionen können individuelle Zugriffsrechte definieren
- **public** - (Standard) erlaubt den Zugriff von außerhalb der Klasse
- **private** - erlaubt nur den Zugriff aus Funktionen der Klasse
- **protected** - erlaubt den Zugriff von innerhalb und erweiterten Klassen

```
class KaffeeAutomat {
    public marke: string;
    public farbe: string;
    private an: boolean;
    constructor(kAMarke, kAFarbe, kAAn) { ... };
    public anschalten = function(): void {
        an = true;
    }
}
```

Vergleich TypeScript - JavaScript

- TypeScript
  - Starke Typisierung möglich
  - Fehlerhafte Verwendung führt zu Fehlermeldung
  - Array-Elemente haben gleichen Typ
  - Tupel als typisierte Arrays
  - Objekte mit typisierten Eigenschaften
  - Funktionen haben typisierte Parameter und Rückgabewerte
  - Klassen erlauben moderne objekt-orientierte Programmierung
- JavaScript
  - Lose Typisierung
  - Fehlerhafte Verwendung wird nicht unmittelbar erkannt
  - Array-Elemente können individuelle Typen haben
  - Eigenschaften von Objekten haben beliebige Typen
  - Parameter und Rückgabewerte von Funktionen haben keine Typen
  - Klassen sind nahezu identisch zu Konstruktoren ohne Klasse