

TypeScript

Einführung und Grundlagen

Einführung und Überblick

TypeScript is a superset of
JavaScript that compiles to clean
JavaScript output.

— Microsoft

JavaScript (JS) is a lightweight
interpreted or JIT-compiled
programming language with
first-class functions.

— Mozilla Developer Network

JavaScript is a prototype-based,
multi-paradigm, dynamic
language, supporting object-
oriented, imperative, and
declarative styles.

— Mozilla Developer Network

JavaScript

- ist eine Programmiersprache
- Code wird in einer Virtual Machine (VM) ausgeführt
- läuft z.B. in *Browsers* oder unter `_Node.js_`
- wird auch in Anwendungen wie z.B. *MongoDB* oder *Adobe Reader* verwendet

Die Gene von JavaScript

Self

Scheme

Java

Objekte

Funktionen

Primitives

Typsystem

Scope

Syntax

Vererbung

Closure

Name

VM

VM

Virtual Machines (VM)

Name	Maintainer	Quelle
V8 ¹	Google	https://chromium.googlesource.com/v8/v8.git
SpiderMonkey [²]	Mozilla	https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey
ChakraCore [³]	Microsoft	https://github.com/Microsoft/ChakraCore
JavaScriptCore [⁴]	Apple	https://developer.apple.com/reference/javascriptcore

¹ V8 is Google's open source JavaScript engine.

[²]: SpiderMonkey is Mozilla's JavaScript engine written in C/C++. It is used in various Mozilla products, including Firefox, and is available under the MPL2.

[³]: ChakraCore is the core part of the Chakra Javascript engine that powers Microsoft Edge

[⁴]: The JavaScriptCore Framework provides the ability to evaluate JavaScript programs from within Swift, Objective-C, and C-based apps.

ECMAScript

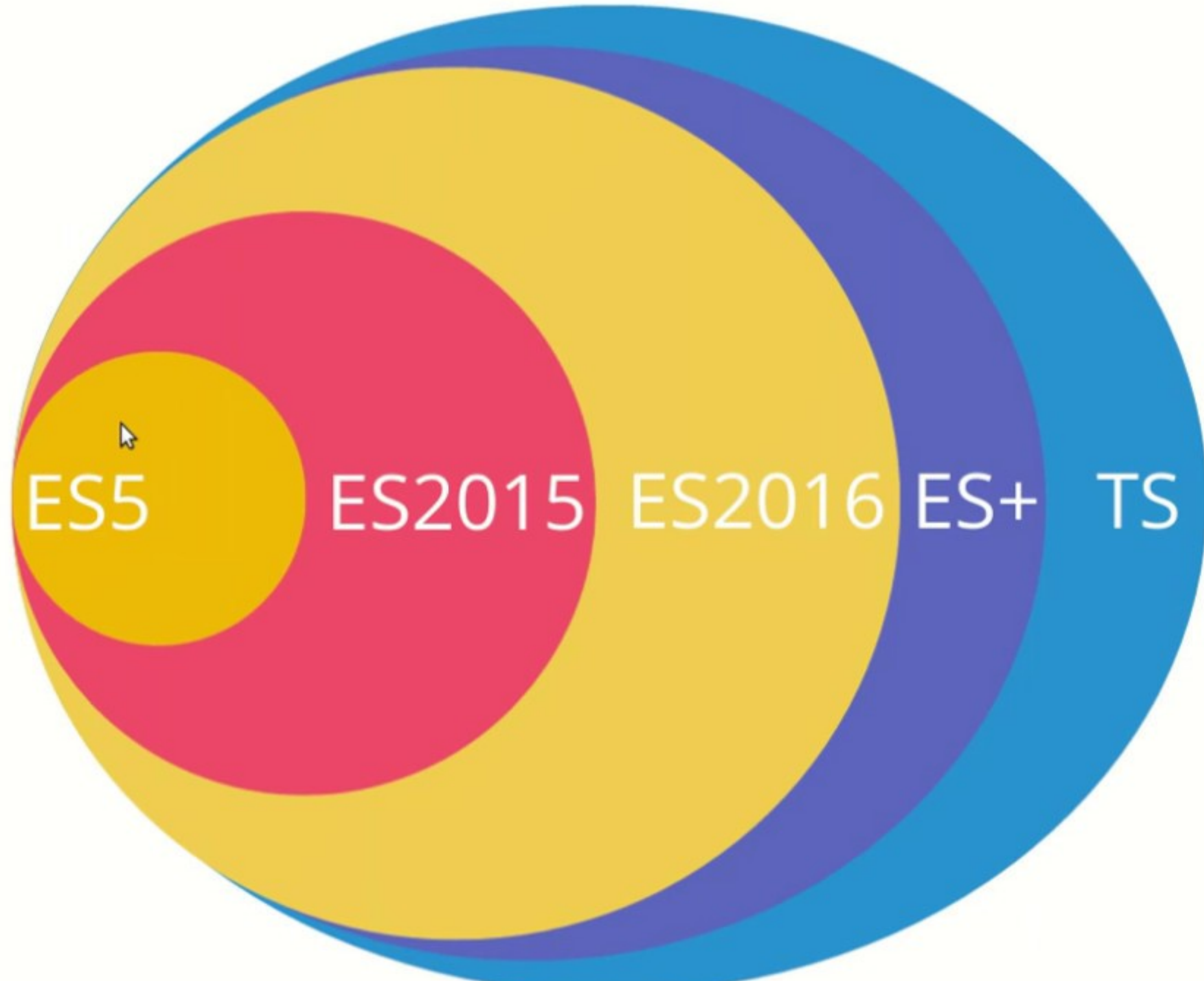
- standardisiert die Eigenschaften von JavaScript
- wurde seit 1996 **unregelmäßig** erweitert (*ES3, ES5*)
- wird seit 2015 **jährlich** erweitert (*ES2015, ES2016, ES2017, ES2018, ES2019*)
- ist von aktuellen VMs *fast vollständig* implementiert ⁵
- erfordert *kontinuierliche Anpassungen* an den VMs

⁵ [ECMAScript compatibility table](#)

TypeScript

- ist eine *Erweiterung von JavaScript*
- ist ein ausgereiftes und aktives *Open Source* Projekt von Microsoft
- wurde "erfunden" von **Anders Hejlsberg**
- liegt aktuell in Version **v3.5.2** vor
- gehört aktuell zu den **“Most Loved Languages”**
- ist in TypeScript programmiert 😊

Jeder valide JavaScript Code ist
automatisch valider TypeScript
Code



Möglichkeiten / Nutzen

- Fehler frühzeitig erkennen
- Neue ECMAScript Features nutzen, um sie in älteres lauffähiges JavaScript zu kompilieren
- Refactoring vereinfachen
- Dokumentation vereinfachen
- Zukunftssicher programmieren
- Apps mit Angular entwickeln

Bausteine (1)

- Compiler (tsc)
 - wandelt TypeScript in JavaScript um
 - warnt vor Fehlern im TypeScript

Bausteine (2)

- Language Server (`tsserver`)
 - wird von `_IDE_s` benutzt, um während der Entwicklung Fehler im Code anzuzeigen

Bausteine (3)

- AST Parser⁶
 - analysiert / transformiert den TypeScript Code
 - wird von den IDEs benutzt, um Refactoring zu vereinfachen

⁶AST Explorer

Werkzeuge

IDE(s)

- Visual Studio Code
- WebStorm
- Atom
- Sublime Text
- Eclipse
- etc.

Weitere

- TypeScript Playground
- `tsc` an der Kommandozeile
- TypeScript execution environment for node
- TypeSearch
- DevDocs
- Quokka.js
- StackBlitz

Syntax, Schlüsselworte

Syntax

- Die grundlegende Syntax folgt weitgehend dem Stil von *Java* und *C*
- TypeScript erweitert JavaScript
- gültiges JavaScript ist immer gültiges TypeScript

Schlüsselworte (1)

- Typ-Annotationen (z.B.)
 - : number
 - : string[]
 - : any
 - : object
 - : Color

Schlüsselworte (2)

- Keywords (z.B.)
 - `type`
 - `interface`
 - `enum`
 - `private`, `public`, `protected`
 - `readonly`
 - `Promise<boolean>`

Basics: Typen, Variablen

Typen in JavaScript (dynamische Typisierung)

- Werte in JavaScript haben zur *Laufzeit* (in der VM) **immer** genau einen Datentyp.
- Variablen beinhalten Werte. Der Wert einer Variablen kann geändert werden, so dass sich auch der Datentyp ändern *kann*.
- Diese *dynamische Typisierung* ist gleichzeitig Stärke wie auch Schwäche von JavaScript

typeof Operator

```
typeof 17; // "number"  
typeof "red"; // "string"  
typeof false; // "boolean"  
typeof undefined; // "undefined"  
typeof {}; // "object"  
typeof []; // "object"  
typeof function() {}; // "function";  
typeof Symbol(); // "symbol"
```

```
let n;
```

```
typeof n; // "undefined"  
n = 17;  
typeof n; // "number"  
n = "red";  
typeof n; // "string"
```

Variablen

- Variablen müssen *deklariert* werden
- Variablen beinhalten Werte
- Werte von Variablen können zur Laufzeit geändert werden
- Werte von Variablen können in andere Variablen kopiert werden

Deklaration mit var, let, const

```
n = 42; // erlaubt
// Variable existiert im function scope
var n; // Deklaration wird gehohistet
```

```
m = 17; // nicht erlaubt
// Variable existiert im block scope
let m;
if (true) {
    let m = 17; // Deklaration wird nicht gehohistet
    typeof m; // "number"
}
typeof m; // "undefined"
```

```
// Variable existiert im block scope
const o = { color: "red" }; // Deklaration und Wertzuweisung müssen ein einem Statement stattfinden

o = 42; // Exception

o.color = "blue"; // erlaubt, das Objekt o ist nicht unveränderbar
```

Grammatik, Deklarationen und Annotationen in TypeScript

Die statische Codeanalyse (*AST Parser*, `tsc`) von TypeScript ermöglicht bereits während der Entwicklungszeit Annahmen über den *Typ* einer *Variablen*.

— Type Inference

Typableitung (1)

```
let m = 17;
```

```
m = "red"; // Warnung
```

```
-----  
let things = [17, "red"]; // Tooltip "(string | number)[]"
```

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

```
const t = typeof multiply(17, 42);
```

```
t; // number
```

Typableitung (2)

```
const thing = {  
  color: "red"  
};
```

```
thing.name = "Test"; // Warnung
```


Statische Typisierung (1)

- Während der Entwicklung von TypeScript-Dateien können *optional* Datentypen angegeben werden
- Dies erfolgt als Annotation mit *:Datentyp*
- Zum Beispiel
 1. Bei der Deklaration von *Variablen*
 2. In *Parametern* einer Funktion
 3. Als *Rückgabetyt* einer Funktion

Statische Typisierung (2)

```
function concat(a: string, b: string): string {  
    return a + b;  
}
```

```
const s: string = concat("Hello", "TypeScript");
```

Statische Typisierung (3)

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;

function getName(n: NameOrResolver): Name {
  if (typeof n === "string") {
    return n;
  } else {
    return n();
  }
}
```

Primitive Datentypen und "any"

Number

- Zahlen werden in JavaScript im **Double-precision floating-point format** verarbeitet
- Datentyp in TypeScript ist `:number` (mit *kleinem* n)
- In Zukunft wird es weitere numerische Datentypen geben, z.B. **BigInt**

```
const pins: number = 8;
```

String (1)

- Strings werden mit `"`, `'` oder ``` umschlossen
- ``` erlaubt
 1. Multiline-Strings
 2. Template-Strings
 3. Tagged Template-Strings
- Strings sind unveränderlich (*immutable*)
- Datentyp in TypeScript ist `: string` (mit *kleinem s*)

String (2)

```
const headline = `<h1>Display Colors</h1>
<h2>An Angular / TypeScript project</h2>
`;

const thing = {
  color: "red"
};

const message: string = `The color of the thing is ${thing.color}!`;
```

Boolean

- Boolsche Werte in JavaScript sind `true` und `false`
- JavaScript kennt das Konzept der *truthy* und *falsy* values **Falsy | MDN**
- Datentyp in TypeScript ist `:boolean` (mit *kleinem* b)

any

- TypeScript kennt zusätzlich den Datentyp : any
- Man kann ihn verwenden
 1. wenn der tatsächliche Datentyp nicht relevant ist
(bitte *mit Vorsicht* benutzen)
 2. wenn man explizit auf den Typ prüfen möchte
- siehe auch die Compiler-Option `--noImplicitAny`
- siehe auch die Dokumentation zu `Advanced Types`

Objekte

— Objekte in JavaScript

1. sind *key / value*-Paare, deren *key* immer ein `string` ist
2. sind *leichtgewichtig* und *dynamisch*
3. sind **nicht** *unveränderbar*
4. erben Eigenschaften *prototypisch* von existierenden Objekten

Objekt Literal

```
let goldenrod = {  
  r: 218,  
  g: 165,  
  b: 32  
};
```

```
goldenrod.a = 1;  
// goldenrod["a"] = 1;  
// const prop = "a";  
// goldenrod[prop] = 1;
```

```
const r = 218;  
const g = 165;  
const b = 32;  
// Verkürzte Form für { r: r, g: g, b: b }  
goldenrod = { r, g, b };
```

Destructuring

```
const goldenrod = {  
  r: 218,  
  g: 165,  
  b: 32  
};  
// Verkürzte Form für const r = goldenrod.r etc.  
const { r, g, b } = goldenrod;  
console.log(r, g, b);
```

Prototypische Vererbung

```
const goldenrod = {
  r: 218,
  g: 165,
  b: 32,
  toString() {
    return `rgba(${this.r}, ${this.g}, ${this.b}${
      this.a ? ", " + this.a : ""
    })`;
  }
};
// Erzeugt ein neues Objekt auf Basis des existierenden Objekts
const dimmed = Object.create(goldenrod);

dimmed.a = 0.5;

console.log(dimmed.toString(), goldenrod.toString()); // rgba(218, 165, 32, .5) rgba(218, 165, 32);

console.log(goldenrod.isPrototypeOf(dimmed)); // true
```

Enumerationen

TypeScript unterstützt zusätzlich Enumerationen.

```
enum LedState {  
    OFF,  
    BLINKING,  
    PERMANENT  
}
```

Funktionen

Eigenschaften

- Funktionen in JavaScript
 1. beinhalten ausführbaren Code
 2. können 0..n *Argumente* entgegennehmen
 3. können 0..1 *Rückgabewerte* haben
 4. können Variablen zugewiesen werden
 5. sind *First-Class-Objekte*

Eine First-Class-Funktion kann ...
als Argument übergeben, ... als
Wert zurückgegeben, einer
Variablen zugewiesen, in einer
Datenstruktur gespeichert und
zur Laufzeit erzeugt werden.

— vgl. Wikipedia

function declaration

- Verwendung des `function` Keyword
- Name der `function` ist *obligatorisch*
- Deklaration und Zuweisung werden *gehoistet*

```
function getColors() {  
    return [  
        {  
            r: 218,  
            g: 165,  
            b: 32  
        }  
    ];  
    // ohne explizites return Statement wird immer undefined zurück gegeben  
}
```

function expression

- Zuweisung zu einer Variablen unter Verwendung der function Expression
- Name der function ist optional
- Deklaration und Zuweisung werden *nicht gehoistet*

```
const getColors = function() {  
  return [  
    {  
      r: 218,  
      g: 165,  
      b: 32  
    }  
  ];  
  // ohne explizites return Statement wird immer undefined zurück gegeben  
};
```

Parameter-Deklarationen

```
// parameter default value
function getRandomColors(count = 1) {
    console.log(count);
}
getRandomColors(); // 1
```

```
// optional parameter
function getRandomColors(count = 1, descending?: boolean) {
    // if (descending) {} else {}
}
```

```
// rest parameter
function getColors(alpha: number, ...hexes: string[]) {
    console.log(Array.isArray(hexes));
}
getColors(0.5, "#fff", "#ccc", "#000"); // true
```

Funktionen höherer Ordnung

Eine Funktion höherer Ordnung (englisch higher-order function) ist in der Informatik eine Funktion, die Funktionen als Argumente erhält oder Funktionen als Ergebnis liefert. Der Begriff wird insbesondere im Lambda-Kalkül verwendet, der theoretischen Grundlage der Funktionalen Programmierung.

— [Wikipedia](#)

Eine Funktion als Funktionsargument

```
function getColors(cb) {  
    cb(null, [  
        {  
            r: 218,  
            g: 165,  
            b: 32  
        }  
    ]);  
}  
// Übergabe einer anonymen Funktion  
getColors(function(err, res) {  
    console.log(err, res);  
});
```

Eine Funktionen als Ergebnis

```
function pick(prop) {  
    return function(obj) {  
        return obj[prop];  
    };  
}
```

```
const pickRed = pick("r");
```

```
typeof pickRed; // "function"
```

```
const pickBlue = pick("b");
```

```
const goldenrod = { r: 218, g: 165, b: 32 };
```

```
pickRed(goldenrod); // 218
```

```
pickBlue(goldenrod); // 165
```


Closure

Eine Closure ... beschreibt eine Funktion, die Zugriffe auf ihren Erstellungskontext enthält. Beim Aufruf greift die Funktion dann auf diesen Erstellungskontext zu. Dieser Kontext ... ist außerhalb der Funktion nicht referenzierbar, d. h. nicht sichtbar.

— [vgl. Wikipedia](#)

"this" als Ausführungskontext

- Innerhalb des `function` Body referenziert das Keyword `this` den **Ausführungskontext der Funktion**
- Der tatsächliche Wert von `this` ist also immer *dynamisch* und kann zur Laufzeit geändert werden.

Arrow-Funktionen / Lambda-Expressions (ES2015)

- Funktionen können auch mit dem "fat arrow" `=>` ("Lambda-Expression") geschrieben werden
- Das Verhalten von `function` und `=>` ist z.T. *unterschiedlich*
- Der `=>` orientiert sich am Konzept des **Lambda-Kalkül**
- TypeScript kompiliert Lambda-Expressions in `function Expressions`

Syntax (1)

```
// implizites return statement
const getColors = () => [{ r: 218, g: 165, b: 32 }];
// exakt 1 parameter, keine () erforderlich
const double = v => v * 2;
// != 1 parameter || typ annotation, () erforderlich
const pick => (prop, obj) => obj[prop];
// explizites return statement || function body, {} erforderlich
const getRandomColors = (count = 1, descending?: boolean) => {
    if (descending) {
        return [].reverse();
    } else {
        return [];
    }
}
```

Syntax (2)

```
// Funktion höherer Ordnung
const pick = (prop: string) => (obj: object) => obj[prop];

const pickRed = pick("r");
typeof pickRed; // "function"

// Funktion höherer Ordnung
const getColors = cb => cb(null, [{ r: 218, g: 165, b: 32 }]);

getColors((err, result: Color[]) => {
    if (err) {
        throw err;
    } else {
        console.log(result);
    }
});
```

Arrow functions have a lexical `this`; its value is determined by the surrounding scope.

— Exploring ES6

Generics

In languages like C# and Java,
one of the main tools ... for
creating reusable components
is ... being able to create a
component that can work over a
variety of types ...

— vgl. TypeScript Handbook

Beispiel "identity"-Funktion

```
// die identity-Funktion gibt das erste an sie übergebene Argument zurück
function identity1(val) {
    return val;
}
let i1 = identity1("red"); // TypeScript nimmt für i1 den Typ any an
i1 = 42; // keine Warnung

// die typisierte Variante
function identity2(val: string): string {
    return val;
}
let i2 = identity2("blue"); // TypeScript nimmt für i1 den Typ string an
let i3 = identity2(42); // Warnung

// die generische Variante
function identity3<T>(val: T): T {
    return val;
}
let i4 = identity3<string>("blue"); // TypeScript nimmt für i4 den Typ string an
let i5 = identity3<number>(42); // TypeScript nimmt für i4 den Typ number an
```

Beispiel Set / Map

```
// ES2015 kennt zusätzlich zu Arrays auch Sets
const colors1 = new Set();
colors1.add("red");
colors1.add({ r: 218, g: 165, b: 32 }); // keine Warnung

// die generische Variante
const colors2 = new Set<string>();
colors2.add("red");
colors2.add({ r: 218, g: 165, b: 32 }); // Warnung

// ... und Maps
const colors3 = new Map<string, object>();
colors3.set("goldenrod", { r: 218, g: 165, b: 32 });
colors3.set(42, "red"); // Warnung
```

Beispiel "Promise"

```
// ES2015 kennt native Promises
Promise.resolve("red").then(res => console.log(res)); // TypeScript nimmt für res den Typ string an
Promise.resolve(42).then(res => console.log(res)); // TypeScript nimmt für res den Typ number an

// generische Variante 1
Promise.resolve<number>("red").then(res => console.log(res)); // Warnung

// generische Variante 2
type Color = { r: number; g: number; b: number };

function getColor(): Promise<Color> {
    return new Promise<Color>((resolve, reject) => {
        resolve({ r: 218, g: 165, b: 32 });
        // resolve("goldenrod"); // Warnung
    });
}
```

Arrays

Arrays in JavaScript

- Arrays in JavaScript
 1. sind *sortierte Listen* von Werten beliebiger Datentypen
 2. können mit Schleifen iteriert werden
 3. haben etliche Methoden für funktionale Programmierung
- Datentyp in TypeScript ist `Array<any>` oder `any[]`

Array Literal

```
const colors: { r: number; g: number; b: number }[] = [  
    { r: 218, g: 165, b: 32 },  
    { r: 255, g: 0, b: 0 },  
    { r: 0, g: 255, b: 0 },  
    { r: 0, g: 0, b: 255 }  
];
```

// Zugriff über index

```
console.log(colors.length); // 4
```

```
console.log(colors[0]); // { r: 218, g: 165, b: 32 }
```

Schleifen

```
// Klassischer for Loop
// kann mit break unterbrochen werden
for (let i = 0; i < colors.length; i++) {
    console.log(i, colors[i]);
}
// ES2015 for ... of, Iterable / Iterator
for (let color of colors) {
    console.log(color);
}
// Funktionale Iteration
// kann nicht mit break unterbrochen werden
colors.forEach(color => console.log(color));
```

Transformation mit map

// map erzeugt ein neues Array aus den Rückgabewerten

```
let reds: number[];
```

```
reds = colors.map(color => {  
    return color.r;  
});
```

```
console.log(reds); // [218, 255, 0, 0]
```

// mit Funktion höherer Ordnung

```
const pick = (prop: string) => (obj: object) => obj[prop];
```

```
reds = colors.map(pick("r"));
```

```
console.log(reds); // [218, 255, 0, 0]
```


Teilmenge mit `filter`

```
// filter erzeugt ein neues Array basierend auf der truthy- / falsyness
let reds: { r: number; g: number; b: number }[];
reds = colors.filter(color => color.r);
console.log(reds); // [{ r: 218, g: 165, b: 32 }, { r: 255, g: 0, b: 0 }]

// mit Funktion höherer Ordnung
const pick = (prop: string) => (obj: object) => obj[prop];
reds = colors.filter(pick("r"));
console.log(reds); // [{ r: 218, g: 165, b: 32 }, { r: 255, g: 0, b: 0 }]
```

Verkettung mit `filter` und `map`

```
// filter und map geben jeweils ein Array zurück
const pick = (prop: string) => (obj: object) => obj[prop];
let reds: number[];

reds = colors.filter(pick("r")).map(pick("r"));

console.log(reds); // [218, 255]
```

Destructuring

```
// Einzelne Variablen können extrahiert werden
const [red, green, blue] = colors;
// red, green, blue sind separate Werte und kein Array
console.log(red, green, blue);
// { r: 255, g: 0, b: 0 }
// { r: 0, g: 255, b: 0 }
// { r: 0, g: 0, b: 255 }
```

Klassen

und

Schnittstellen

Klassen

Klassen definieren

- Klassen sind **eine verkürzte Schreibweise** für prototypische Vererbung ("syntactic sugar")
- Nicht zu verwechseln mit Klassen in *Java*, *C++*, *C#* etc.
- ES2015: `class`, `constructor`, `extends`, `super` und `static` (für *Funktionen*)
- TypeScript: `public`, `private`, `protected`, `readonly`, `abstract` und `static` (für *Properties*)

— Zuweisung der property-Werte in der constructor-Funktion

```
class Color {  
  constructor(r, g, b, a = 1) {  
    this.r = r;  
    this.g = g;  
    this.b = b;  
    this.a = a;  
  }  
}
```

```
const c = new Color(255, 255, 127, 0.5);
```

TypeScript Klasse (1)

Deklaration der properties und - *optional* - ihrer Datentypen

TypeScript Klasse (2)

```
class Color {  
  r: number;  
  g: number;  
  b: number;  
  a: number;  
  constructor(r: number, g: number, b: number, a: number = 1) {  
    this.r = r;  
    this.g = g;  
    this.b = b;  
    this.a = a;  
  }  
}
```

```
const c = new Color(255, 255, 127, 0.5);
```

Modifier im Constructor (1)

- Verkürzte Schreibweise erspart die Zuweisung der property-Werte in der `constructor`-Funktion
- Wird außerdem in Angular für *Dependency Injection* genutzt

Modifier im Constructor (2)

```
class Color {  
    constructor(  
        public r: number,  
        public g: number,  
        public b: number,  
        public a: number = 1  
    ) {}  
}
```

```
const c = new Color(255, 255, 127);
```

Funktionen in Klassen (1)

- a.k.a. *Methoden*
- Verkürzte Schreibweise ohne `function` Keyword
- `this` referenziert auf das Objekt, welches mit `new` erzeugt wurde
- ES5 `get` und `set` werden unterstützt
- TypeScript unterstützt zusätzlich *Typ-Annotationen* und *Zugriffs-Modifizier*

Funktionen in Klassen (2)

```
class Color {
  constructor(
    public r: number,
    public g: number,
    public b: number,
    public a: number = 1
  ) {}

  toString() {
    return `rgba(${this.r}, ${this.g}, ${this.b}${
      this.a ? ", " + this.a : ""
    })`;
  }

  get red(): number {
    return this.r;
  }

  set red(value: number) {
    this.r = value;
  }
}

const c = new Color(128, 165, 32);
c.red = 255;
console.log(c.toString(), c.red);
```

static und readonly in TypeScript

```
class Color {
    static readonly MIN = 0;
    static readonly MAX = 255;

    constructor(
        public r: number,
        public g: number,
        public b: number,
        public a: number = 1
    ) {}

    static create({
        r,
        g,
        b,
        a
    }: {
        r: number;
        g: number;
        b: number;
        a?: number;
    }): Color {
        return new Color(r, g, b, a);
    }
}
```

Nutzung der *factory*-Funktion `create`

```
const goldenrod: Color = Color.create({  
  r: 218,  
  g: 165,  
  b: 32  
});
```

```
const black: Color = Color.create({  
  r: Color.MIN,  
  g: Color.MIN,  
  b: Color.MIN  
});
```

```
const white: Color = Color.create({  
  r: Color.MAX,  
  g: Color.MAX,  
  b: Color.MAX  
});
```

Datentypen erweitern

Vererbung (1)

- Vererbung ist möglich, sollte aber **sehr flach** gehalten werden
- ES2015 unterstützt `extends` und `super`
- TypeScript unterstützt zusätzlich `public`, `private`, `protected` und `abstract`

Vererbung (2)

```
export class BlinkingColor extends Color {
  constructor(
    r: number,
    g: number,
    b: number,
    a: number,
    private frequency = 1
  ) {
    super(r, g, b, a);
  }

  blink(duration: number) {
    return `${this.toString()} is blinking with ${
      this.frequency
    } Hz for ${duration} ms`;
  }
}
```

Zugriffs-Modifier

- `public`: *auch außerhalb* der Klasse erreichbar
- `private`: *ausschließlich innerhalb* der Klasse erreichbar
- `protected`: nur *innerhalb* der Klasse sowie *innerhalb von erbenden* Klassen erreichbar
- `abstract`: kann nicht mit `new` instantiiert werden

Schnittstellen

Schnittstellen verwenden

- ECMAScript kennt **keine** Schnittstellen
- TypeScript unterstützt *Schnittstellen* für die **Beschreibung** von Objekten und Klassen
- Interfaces * TypeScript

Strukturelles Interface

- `interface`: beschreibt eine *Struktur*
- Schnittstellen können beliebig tief *verschachtelt* sein

```
interface IRGB {  
    r: number;  
    g: number;  
    b: number;  
}
```

Verschachtelung

```
interface ILed {  
  index: number;  
  color: {  
    r: number;  
    g: number;  
    b: number;  
  };  
}  
// color: IRGB  
  
const led: ILed = {  
  index: 0,  
  color: {  
    r: 218,  
    g: 165,  
    b: 32  
  }  
};
```

Vererbung

- extends: Schnittstellen unterstützen *Vererbung*
- ?: markiert *optionale Properties*

```
interface IRGBA extends IRGB {  
    a?: number;  
}
```


Funktionen in Schnittstellen (1)

- Schnittstellen können *Properties* beschreiben, die *Funktionen* sind
- Schnittstellen können *Funktionen* beschreiben

Funktionen in Schnittstellen (2)

```
interface IColor {  
    toString(): string;  
    red: number;  
}
```

```
interface BlinkFunc {  
    (duration: number): string;  
}
```

```
interface IBlinkingColor extends IColor {  
    blink: BlinkFunc;  
}
```

Verwendung von Schnittstellen (1)

- Variablen können Schnittstellen als *Datentyp* haben
- `implements`: Klassen können Schnittstellen implementieren
- Schnittstellen können in Signaturen von *Funktionen* verwendet werden

Verwendung von Schnittstellen (2)

```
class Color implements IRGBA, IColor {
  constructor(
    public r: number,
    public g: number,
    public b: number,
    public a: number = 1
  ) {}

  toString() {
    return `rgba(${this.r}, ${this.g}, ${this.b}${
      this.a ? ", " + this.a : ""
    })`;
  }

  get red(): number {
    return this.r;
  }

  set red(value: number) {
    this.r = value;
  }

  static create({ r, g, b, a }: IRGBA): IColor {
    return new Color(r, g, b, a);
  }
}
```

Ambiente Deklarationen und Type Declaration Files

Externe Bibliotheken einbinden

In JavaScript geschriebene Bibliotheken können problemlos in TypeScript benutzt werden

```
npm install tinycolor2 --save
```

```
import * as tinycolor from "tinycolor2";
```

```
const goldenrod = tinycolor("goldenrod");
```

```
// Hier kennt TypeScript den Datentyp von goldenrod noch nicht
```

Type Deklarationen `.d.ts` via `@types` installieren

Für sehr viele Bibliotheken existieren Definitionen der Datentypen für TypeScript

```
npm install @types/tinycolor2 --save-dev
```

```
console.log(tinycolor("goldenrod").isDark()); // false
```

```
// Hier kennt TypeScript alle Eigenschaften und Methoden von goldenrod
```

This guide is designed to teach
you how to write a high-quality
TypeScript Declaration File.

— TypeScript Handbook

Fazit

TypeScript ranks 3rd in the “Most Loved Languages” category — even ahead of Swift, Go, and Python! — and 9th in the “Most Popular Languages” category.

— TypeScript Weekly

Anhang

Hello TypeScript

Voraussetzungen schaffen

1. **Node.js** installieren
2. **Visual Studio Code** installieren (oder vergl. Editor)
3. *optional* **git** installieren
4. *optional* **Yarn** installieren

Projekt initialisieren

1. Neues Verzeichnis anlegen:

```
mkdir my-typescript-project && cd -
```

2. Im neuen Verzeichnis Node.js initialisieren:

```
npm init -y oder yarn init -y
```

3. Notwendige Node.js Pakete installieren:

```
npm install --save-dev typescript oder  
yarn add typescript -D
```

TypeScript verwenden

1. TypeScript initialisieren:
`./node_modules/.bin/tsc --init`
2. Compiler konfigurieren in `tsconfig.json`
3. Datei erstellen `hello.ts`
4. Programmieren :-)
5. Compiler ausführen `./node_modules/.bin/tsc`

Unit-Tests konfigurieren

- Jest
- ts-jest