

# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 1 von 2

## Variablen und ihre Gültigkeitsbereiche

**"use strict";** // JavaScript im strengen Modus (keine undeklarierten Variablen erlaubt)  
// console.log(y); // kein Zugriff vor Initialization, da im Strict Mode!  
// console.log(z); // not defined, da im Strict Mode!

```
var a = "Oliver Sturm"; // var sollte man nicht mehr verwenden, da es zu unerwarteten Ergebnissen führen kann. "var" ist global gültig, auch wenn es in einem Block definiert wird.  
let b = 49; // let nur verwenden, wenn die Variable sich tatsächlich ändern kann. "let" ist nur in dem Block gültig, in dem es definiert wurde.  
const c = "Hallo"; // verwenden, wann immer möglich  
{  
  var a = "Holger Schwichtenberg"; // überschreibt globales a  
  let b = 50; // neues b mit Gültigkeit nur in diesem Block  
  const c = "Hello"; // neues c mit Gültigkeit in diesem Block  
  console.log(a); // Holger Schwichtenberg  
  console.log(b); // 50  
  console.log(c + " " + a); // Hello Holger Schwichtenberg  
}  
console.log(a); // Holger Schwichtenberg  
console.log(b); // 49  
console.log(c + " " + a); // Hallo Holger Schwichtenberg
```

```
let x = 41; // eine Zahl  
const y = x + 1; // Konstante darf Ausdruck enthalten  
// y++; // Fehler: Konstante darf nicht verändert werden
```

## Datentypen und Literale

```
const eineZahl = 42;  
const eineAndereZahl = 42.5; // selber Typ wie eineZahl  
const eineGrosseZahl = 42_000_000_000.123; // selber Typ wie eineZahl  
const einString = "Oli"; // Bei "..." keine Interpolation erlaubt  
const einWeitererString = 'Holger'; // Bei '...' keine Interpolation erlaubt  
const einStringTemplate = `eineZahl = ${eineZahl}`; // `...` erlaubt Interpolation  
const einWeiteresStringTemplate = `Interpolation kann beliebige Ausdrücke enthalten: ${eineZahl + 1}`;  
const einBoolean = true;  
const einRegExp = /abc/;  
const einDatum = new Date(2023, 7, 15);
```

// Typen von Werten

```
console.log(`eineZahl=${eineZahl} Typ=${typeof eineZahl}`); // number  
console.log(`einString=${einString} Typ=${typeof einString}`); // string  
console.log(`einBoolean=${einBoolean} Typ=${typeof einBoolean}`); // boolean  
console.log(`einRegExp=${einRegExp} Typ=${typeof einRegExp}`); // object  
console.log(`einDatum=${einDatum} Typ=${typeof einDatum}`); // object
```

```
if (typeof eineZahl === "number") {  
  console.log("eineZahl ist eine Zahl!");  
}
```

```
if (isNaN(einString)) {  
  console.log("n enthält keine Zahl!");  
}
```

## Operatoren

```
console.log(1 + 2); // 3  
console.log(1 - 2); // -1  
console.log(1 * 2); // 2  
console.log(1 / 2); // 0.5  
console.log(1 % 2); // 1 (Modulus)  
console.log(2 ** 3); // 8 (Exponentiation)  
console.log(1 > 2); // false  
console.log(1 < 2); // true  
console.log(1 >= 2); // false  
console.log(1 <= 2); // true  
console.log(1 == 2); // false (gleich, mit Typumwandlung)  
console.log(1 === 2); // false (gleich, ohne Typumwandlung)  
console.log(1 != 2); // true (ungleich, mit Typumwandlung)  
console.log(1 !== 2); // true (ungleich, ohne Typumwandlung)  
console.log(1 && 2); // 2 (logisches Und, Kurzschlussauswertung)  
console.log(1 || 2); // 1 (logisches Oder, Kurzschlussauswertung)  
console.log(!1); // false (logisches Nicht)  
console.log(1 & 2); // 0 (bitweises Und)  
console.log(1 | 2); // 3 (bitweises Oder)  
console.log(1 ^ 2); // 3 (bitweises Exklusiv-Oder)  
console.log(~1); // -2 (bitweises Nicht)  
console.log(1 << 2); // 4 (bitweises Linksverschieben)  
console.log(1 >> 2); // 0 (bitweises Rechtsverschieben mit Vorzeichen)  
console.log(1 >>> 2); // 0 (bitweises Rechtsverschieben ohne Vorzeichen)  
const koennteUndefinedSein = undefined;  
console.log(koennteUndefinedSein ?? "Standardwert"); // Standardwert  
console.log(1 ? 2 : 3); // 2
```

## modifizierende Operatoren

```
+ =, - =, * =, / =, % =, ** =, < < =, > > =, > > > =, & =, ^ =, | =  
let z = 10;  
z *= 4;  
z += 1;  
z++;  
console.log(z); // 42
```

## Bedingungen mit if und switch

```
var z1 = Math.round(Math.random() * (10 - 1)) + 1;  
if (z1 > 5) {  
  console.log("Die Zahl " + z1 + " ist größer als 5");  
} else if (z1 == 5) {  
  console.log("Die Zahl ist gleich 5");  
} else {  
  console.log("Die Zahl " + z1 + " ist größer 5");  
}
```

```
switch (z1) {  
  case 1:  
    console.log("Die Zahl ist 1");  
    break;  
  case 2:  
    console.log("Die Zahl ist 2");  
    break;  
  case 3:  
    console.log("Die Zahl ist 3");  
    break;  
  default:  
    console.log("Die Zahl ist größer als 3");  
    break;  
}
```

## == vs. ===

**== nur nutzen, wenn explizit dynamische Typisierung gewünscht ist!**

```
const v1 = 1;  
const v2 = "1";  
const v3 = true;
```

**Vergleiche mit == und != → alle melden: 'gleich'**

```
if (v1 == v2) { console.log("v1 gleich v2"); }  
if (v1 != v2) { console.log("v1 ungleich v2"); }  
if (v2 == v3) { console.log("v2 gleich v3"); }  
if (v2 != v3) { console.log("v2 ungleich v3"); }  
if (v1 == v3) { console.log("v1 gleich v3"); }  
if (v1 != v3) { console.log("v1 ungleich v3"); }
```

**Vergleiche mit === und !== → alle melden: 'ungleich'**

```
if (v1 === v2) { console.log("v1 gleich v2"); }  
if (v1 !== v2) { console.log("v1 ungleich v2"); }  
if (v2 === v3) { console.log("v2 gleich v3"); }  
if (v2 !== v3) { console.log("v2 ungleich v3"); }
```

# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 2 von 2

```
if (v1 === v3) { console.log("v1 gleich v3"); }
if (v1 !== v3) { console.log("v1 ungleich v3"); }
```

## Truthy und Falsy

**Truthy** – alles, was zu *true* ausgewertet wird

```
console.log(`!!true ist ${!!true}`);
console.log(`!!{} (leeres Objekt) ist ${!!{}}`);
console.log(`!![] (leeres Array) ist ${!![]}`);
console.log(`!!(function(){} (leere Funktion) ist ${!!function () {}}`);
console.log(`!!42 ist ${!!42}`);
console.log(`!!'xy' (nicht-leere Zeichenkette) ist ${!!'xy'}`);
console.log(`!!-42 ist ${!!-42}`);
console.log(`!!Infinity ist ${!!Infinity}`);
```

**Falsy** – alles, was zu *false* ausgewertet wird

```
const obj = new Object();
console.log(`!!false ist ${!!false}`);
console.log(`!!null ist ${!!null}`);
console.log(`!!undefined ist ${!!undefined}`);
console.log(`!!obj ist ${!!obj}`);
console.log(`!!0 ist ${!!0}`);
console.log(`!!NaN ist ${!!NaN}`);
console.log(`!!" (leere Zeichenkette) ist ${!!""}`);
```

**!! --> "cast" zu Boolean**

```
console.log(!!"Irgendein String"); // true
console.log(!!42); // true
console.log(!!""); // false
console.log(!!0); // false
```

## Schleifen

**for-Schleife vorwärts**

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

**for-Schleife rückwärts**

```
for (let i = 10; i >= 1; i--) {
  console.log(i);
}
```

**Kopfgeprüfte while-Schleife**

```
let i = 1;
while (i <= 10) {
  console.log(i);
  i++;
}
```

**Fußgeprüfte while-Schleife**

```
let k = 1;
do {
  console.log(k);
  k++;
} while (k <= 10);
```

**Endlosschleife mit Abbruchbedingung**

```
let m = 1;
do {
  console.log(m);
  if (m >= 10) break;
  m++;
} while (true);
```

**for-of iteriert über die Elemente eines Arrays**

```
for (const val of [1, 2, 3]) {
  console.log(val);
}
```

**for-in iteriert über die Eigenschaften eines Objekts**

```
const objektMitWerten = { a: 1, b: 2, c: 3 };
for (const val in objektMitWerten) {
  console.log(`Objekt-Eigenschaft ${val} hat den Wert ${objektMitWerten[val]}`);
}
```

## Arrays

**Array erzeugen**

```
const zufallszahlen = [];
for (let i = 0; i < 10; i++) {
  zufallszahlen.push(Math.round(Math.random() * (100 - 1)) + 1);
  // oder: zufallszahlen[i] = Math.round(Math.random() * (100 - 1)) + 1;
}
```

**Iteration über Array mit forEach (kann nicht abgebrochen werden)**

```
zufallszahlen.forEach((element) => console.log(element));
```

**Iteration über Array mit for...of (mit Abbruchbedingung)**

```
for (const element of zufallszahlen) {
  console.log(element);
  if (element === 42) break;
}
```

**Verarbeitung von Arrayinhalten mit Standardfunktionen**

```
const zufallszahlenGroesser50 = zufallszahlen.filter((wert) => wert > 50);
const erste10GrosseZufallszahlen = zufallszahlenGroesser50.slice(0, 10);
const quadratZahlen = erste10GrosseZufallszahlen.map((wert) => wert * wert);
const summe = quadratZahlen.reduce((summe, wert) => summe + wert, 0);
```

**Spread Syntax**

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

```
const numbers = [42, 49, 50];
console.log(calc(...numbers)); // 141
```

**Pattern "unveränderbare Daten"**

```
const strings1 = ["Oli", "Holger"];
```

**Pattern "unveränderbare Daten": "Anhängen" eines neuen Elements**

```
const strings2 = strings1.concat("JavaScript");
```

**Alternative: Spread Syntax**

```
const strings3 = [...strings1, "JavaScript"];
```

## Typisierte, leistungsoptimierte Zahlenarrays

**Array erzeugen**

```
const zufallszahlen32 = new Float32Array(10);
for (let i = 0; i < 10; i++) {
  // kein push() vorhanden zufallszahlen32.push...;
  zufallszahlen32[i] = Math.random() * (100_000_000 - 1) + 1;
}
```

**Iteration über Array mit forEach**

```
zufallszahlen32.forEach((element) => console.log(element));
```

**Iteration über Array mit for...of**

```
for (const element of zufallszahlen32) {
  console.log(element);
}
```

**Verfügbare Arraytypen: Int8Array, Int16Array, Int32Array, Uint8Array, Uint8ClampedArray, Uint16Array, Uint32Array, Float32Array, Float64Array, BigInt64Array, BigUint64Array**

## Sets (Mengen ohne Duplikate)

```
const coronaJahre = new Set();
coronaJahre.add(2020);
coronaJahre.add(2021);
coronaJahre.add(2022);
console.log(`${coronaJahre.size} Jahreszahlen im Set!`); // 3
coronaJahre.add(2022);
console.log(`${coronaJahre.size} Jahreszahlen im Set!`); // immer noch 3, da keine Duplikate im Set möglich!
```

**Iteration über Set mit forEach**

```
coronaJahre.forEach((element) => console.log(element));
```

**Iteration über Set mit for...of**

```
for (const element of coronaJahre) {
  console.log(element);
}
```

## Maps (Name-Wert-Paare)

```
const autoren = new Map();
autoren.set("Oliver Sturm", 49);
autoren.set("Holger Schwichtenberg", 50);
console.log(autoren.get("Oliver Sturm")); // 49
for (const [key, value] of autoren) {
  console.log(key + " ist " + value + " Jahre alt.");
}
autoren.delete("Holger Schwichtenberg");
console.log(autoren.size + " Elemente in der Map");
```

# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 3 von 2

## WeakMaps

*WeakMaps* enthalten "schwache" Referenzen auf Objekte. Sollte das Objekt nicht mehr anderswo referenziert werden, kann es vom Garbage Collector entfernt werden.

```
const wmap = new WeakMap();
```

### Eindeutige Objekte als Schlüssel

```
const schluessel1 = { name: "Thema1" };  
const schluessel2 = Symbol("Thema2"); // Symbol erfordert Node.js  
20 oder höher! Browser: siehe: https://caniuse.com/mdn-javascript\_builtins\_weakmap\_symbol\_as\_keys
```

### Wert setzen oder hinzufügen

```
wmap.set(schluessel1, "JavaScript");
```

### Gibt es einen Wert für den Schlüssel?

```
console.log(wmap.has(schluessel1)); // true  
console.log(wmap.has(schluessel2)); // false
```

### Wert holen

```
console.log(schluessel1.name + "=" + wmap.get(schluessel1)); //  
Thema1=JavaScript  
console.log(schluessel2.description + "=" +  
wmap.get(schluessel2)); // Thema2=undefined
```

### // Wert entfernen

```
wmap.delete(schluessel1);
```

## Funktionen und Lambda-Ausdrücke

```
function addiere1(zahl1, zahl2, zahl3) {  
  if (zahl1 === undefined) {  
    throw new Error("Es fehlt zahl1");  
  }  
  if (zahl2 === undefined) {  
    throw new Error("Es fehlt zahl2");  
  }  
  if (zahl3 === undefined) {  
    throw new Error("Es fehlt zahl3");  
  }  
  return zahl1 + zahl2 + zahl3;  
}  
console.log(addiere1(30, 10, 2));
```

```
try {  
  console.log(addiere1(30, 10));  
}  
catch (error) {  
  console.warn("FEHLER: " + error.message);  
}
```

### Funktion via Lambda (Arrow function)

```
const addiere2 = (a, b) => a + b;  
console.log(addiere2(40, 2));
```

### Funktionen ohne Namen

```
const addiere3 = function (a, b) {  
  return a + b;  
};  
console.log(addiere3(40, 2));
```

### Funktionen in funktionaler (Curry) Syntax

```
const addiere4 = (a) => (b) => a + b;  
console.log(addiere4(40)(2));
```

## Funktionen als Parameter

### Funktion ohne Callback

```
function calc1(zahl1, zahl2, zahl3) {  
  return (zahl1 + zahl2) * zahl3;  
}
```

### Funktion mit Callback

```
function calc2(zahl1, zahl2, zahl3, callback) {  
  callback((zahl1 + zahl2) * zahl3);  
}
```

### Funktion mit optionalem Callback

```
function print(inhalt) {  
  if (typeof inhalt === "function") {  
    console.log(inhalt());  
  } else {  
    console.log(inhalt);  
  }  
}
```

### Aufruf mit Wert als Parameter

```
print(calc1(20, 1, 2));
```

### Aufruf mit Funktion als Parameter via Arrow function

```
print(() => calc1(20, 1, 2));
```

### Aufruf mit Funktion als Parameter via Arrow function

```
const f = () => calc1(20, 1, 2);  
print(f);
```

### Funktion als Parameter

```
calc2(20, 1, 2, print);
```

### Arrow function als Parameter

```
calc2(20, 1, 2, (text) => console.log(text));
```

### Funktionen als Rückgabewert

```
function erzeugeCalc(wertZumAddieren) {  
  return (zahl) => zahl * wertZumAddieren;  
}
```

```
const calc3 = erzeugeCalc(2);  
console.log(calc3(21));
```

## Immediately Invoked Function Expression (IIFE)

```
(function () {  
  console.log("Hallo JavaScript");  
})();
```

```
const berechnet = ((kontext) => {  
  return kontext.initWert + 1;  
})({ initWert: 41 });
```

```
console.log(berechnet);
```

## Asynchrone Funktionen

```
function calcMitZeitbegrenzung(x, y) {  
  return new Promise((resolve, reject) => {  
    let z = 6;  
    // alle 200ms ein Rechenschritt  
    const i = setInterval(() => {  
      z = z + x + y;  
      if (z > 100) {  
        clearInterval(i);  
        reject("Ergebnis ist zu groß");  
      }  
      console.log(`Zwischenergebnis um ${new Date().toISOString()}:  
${z}`);  
    }, 200);  
    // nach 2sec abbrechen und das Ergebnis zurückgeben  
    setTimeout(() => {  
      clearInterval(i);  
      resolve(z);  
    }, 2000);  
  });  
}
```

```
function asynchronerAufrufMitThen() {  
  console.log("Asynchrone Berechnung beginnt");  
  calcMitZeitbegrenzung(1, 3).then((result) => {  
    console.log("Endergebnis: " + result); // 42  
  });  
  calcMitZeitbegrenzung(1, 57).then((result) => {  
    console.log("Endergebnis: " + result); // 42  
  }).catch((error) => {  
    console.error("Fehler: " + error);  
  });  
}
```

```
async function asynchronerAufrufMitAsyncAwait() {
```

# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 4 von 2

```
console.log("Asynchrone Berechnung beginnt");
const result = await calcMitZeitbegrenzung(1, 3);
console.log("Endergebnis: " + result); // 42
```

```
try {
  const result = await calcMitZeitbegrenzung(1, 3);
  console.log("Endergebnis: " + result); // 42
}
catch (error) {
  console.error("Fehler: " + error);
}
```

```
asynchronerAufrufMitThen();
asynchronerAufrufMitAsyncAwait();
```

## Generatoren

Ein Generator (mit Stern nach function) kann Werte verschiedener Typen zurückgeben

```
function* verschiedeneWerte() {
  yield 42;
  yield true;
  yield "Hello";
  yield 1.2;
  yield false;
  yield new Date();
}
```

```
for (const zahl of verschiedeneWerte()) { console.log(zahl); }
```

Dieser Generator filtert die Zahlen heraus

```
function* nurZahlen(werte) {
  for (const wert of werte) {
    if (typeof wert === "number") {
      yield wert;
    }
  }
}
```

```
for (const zahl of nurZahlen(verschiedeneWerte())) {
  console.log(zahl);
}
```

Dieser Generator endet nie

```
function* unendlichVieleWerte() {
  let i = 0;
  while (true) {
    yield i++;
  }
}
```

Sequenzfunktion, liefert Werte <n

```
function* take(werte, n) {
  let i = 0;
  for (const wert of werte) {
    if (i >= n) {
      break;
    }
    yield wert;
    i++;
  }
}
```

```
for (const zahl of take(unendlichVieleWerte(), 10)) {
  console.log(zahl);
}
```

## Asynchrone Generatoren und Iteratoren

Ein Generator kann asynchron arbeiten

```
async function* verschiedeneAsyncWerte() {
  yield 42;
  yield true;
  yield "Hello";
}
```

for-await-of ist zur Verarbeitung asynchroner Iteratoren notwendig

```
(async () => {
  for await (const wert of verschiedeneAsyncWerte()) {
    console.log(wert);
  }
})();
```

## Fehlerbehandlung mit try...catch

```
function addiere(zahl1, zahl2) {
  if (zahl1 === undefined) {
    throw new Error("Es fehlt zahl1");
  }
  if (zahl2 === undefined) {
    throw new Error("Es fehlt zahl2");
  }
  return zahl1 + zahl2;
}
console.log(addiere(40, 2));
```

```
let x = 0;
try {
  x = addiere(40);
} catch (error) {
  console.warn("FEHLER: " + error.message);
  x = -1;
}
```

```
} finally {
  if (x < 0) console.log("Es gab leider kein Ergebnis :-(");
  else console.log(`Ergebnis: ${x}`);
}
```

## Fehlerbehandlung bei Promises

```
const name = "Oliver Sturm";
const databaseValue = connectDatabase().then((connection) =>
  connection
  .query(`select id from users where name="${name}"`)
  .then(({ id }) => id)
  .catch((err) => {
    console.error("Error while querying the database", err);
  })
  .finally(() => {
    connection.close();
  })
);
```

## Objekte

```
const p1 = { name: "Holger", alter: 49, geschlecht: "m" };
p1.alter++;
const p2 = { name: "Holger", alter: 50, geschlecht: "m" };
```

Neues Objekt mit Spread-Operator beim Pattern "unveränderbare Daten"

```
const p3 = { ...p1, alter: p1.alter + 1 };
```

Objektreferenzvergleich

```
console.log(p1 === p2); // false
```

Objektinhaltsvergleich

```
console.log(JSON.stringify(p1) === JSON.stringify(p2)); // true
```

Objekt dynamisch um eine Eigenschaft erweitern

```
p2.ort = "Essen";
console.log(`${p2.name} wohnt in ${p2.ort}`);
```

Alle Eigenschaften eines Objekts ausgeben

```
for (const property in p2) {
  console.log(`${property}: ${p2[property]}`);
}
```

JSON-Serialisierung und Deserialisierung

```
const json = JSON.stringify(p2);
console.log(json);
const p3 = JSON.parse(json);
console.log(`${p3.name} wohnt in ${p3.ort}`);
```

# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 5 von 2

## Klassen und Vererbung

```
class Person {
  constructor(vorname, name) {
    this.name = name;
    this.vorname = vorname;
    Person.#instanzZaehler++;
    this.#id = Person.#instanzZaehler;
    Person.alleInstanzen.push(this);
  }

  static alleInstanzen = []; // statisches öffentliches Mitglied
  static #instanzZaehler = 0; // statisches privates Klassenmitglied
  #id = 0; // privates Instanzmitglied (Field)

  get ganzerName() {
    return `${this.vorname} ${this.name}`;
  }

  set ganzerName(ganzername) {
    const namensTeile = ganzername.split(" ");
    if (namensTeile.length !== 2) throw new Error("Ungültiger Name");
    this.vorname = namensTeile[0];
    this.name = namensTeile[1];
  }

  print() {
    console.log(`Person #${this.#id}: ${this.ganzerName}`);
  }
}

// Instanz erzeugen
const p = new Person("Dr. Holger", "Schwichtenberg");
console.log(p.ganzerName);
// console.log(p.#id); // Zugriff auf privates Mitglied nicht möglich
p.print();

// Vererbung
class Trainer extends Person {
  constructor(vorname, name, fach) {
    super(vorname, name);
    this.fach = fach;
  }
}

// Überschreiben von Methoden
print() {
  console.log(`Trainer ${this.ganzerName} unterrichtet
  ${this.fach}`);
}
```

```
const t = new Trainer("Oliver", "Sturm", "JavaScript");
t.ganzerName = "Oli Sturm";
t.print();
```

```
console.log("Es gibt nun " + Person.alleInstanzen.length + "
Personen.");
```

## this

Methoden in Klassen sind standardmäßig gebunden, aber generell wird "this" zur Laufzeit gebunden und kann "von außen" definiert werden.

```
class Ding {
  constructor() {
    this.name = "Ding";
  }

  print() {
    if (this) console.log(this.name);
    else console.log("'this' ist nicht definiert");
  }
}

const d = new Ding();
d.print(); // "Ding"

const kopieVonPrint = d.print;
kopieVonPrint(); // nicht definiert

const gebundeneKopieVonPrint = d.print.bind(d);
gebundeneKopieVonPrint(); // "Ding"

const externesPrint = function () {
  if (this) this.print();
  else console.log("'this' ist nicht definiert");
}.bind(d);
externesPrint(); // "Ding"

// Arrow Functions haben kein eigenes "this", sondern binden "this"
an den Kontext, in dem sie definiert sind.
const externesPrintArrowFunction = () => {
  if (this.print) this.print();
  else console.log("'this' enthält kein 'print'");
}.bind(d);
externesPrintArrowFunction(); // nicht definiert
```

```
class DingMitArrowFunction {
  constructor() {
    this.name = "Ding mit arrow function";
  }
}
```

```
this.print = () => {
  if (this) console.log(this.name);
  else console.log("'this' ist nicht definiert");
};
}
```

```
const d2 = new DingMitArrowFunction();
d2.print(); // "Ding mit arrow function"
```

```
const kopieVonPrint2 = d2.print;
kopieVonPrint2(); // "Ding mit arrow function"
```

```
const externesPrint2 = function () {
  if (this) this.print();
  else console.log("'this' ist nicht definiert");
}.bind(d2);
```

```
externesPrint2(); // "Ding mit arrow function"
```

## WeakRef

Es gibt ein Objekt, zu dem die Variable "oli" eine Referenz hält.

```
let oli = new Person("Oliver", "Sturm");
```

An anderer Stelle gibt es ebenfalls eine Referenz, aber diese wird mittels WeakRef gehalten.

```
const weakPersonRef = new WeakRef(oli);
```

```
const checkRef = (r) => {
  console.log(`WeakRef ist definiert: ${r.deref() !== undefined}`);
};
```

Natürlich ist die Referenz derzeit definiert.

```
checkRef(weakPersonRef); // true
```

Mit dem Objekt kann "ganz normal" gearbeitet werden.

```
oli.ganzerName = "Oli Sturm";
```

An der zweiten Referenz ändert sich dadurch nichts.

```
checkRef(weakPersonRef); // true
```

Nun wird die "Hauptreferenz" irgendwann nicht mehr benötigt.

Das Objekt wird für die Garbage Collection freigegeben.

```
oli = null;
```

Die zweite, schwache, Referenz wird entfernt, wenn die Garbage Collection durchgeführt wird. Dieser Zeitpunkt ist nicht vorhersehbar. Bei Aufruf dieses Tests mit "node --expose-gc JavaScript-ObjekteUndKlassen.js" führt der folgende Block im Allgemeinen zu einer Ausgabe von "false" (aber keine Garantie!).



# Spickzettel („Cheat Sheet“): Modernes JavaScript

Autoren: Dr. Holger Schwichtenberg und Oliver Sturm

V1.4.1 / 22.09.2023 / Seite 6 von 2

```
setTimeout(() => {  
  global.gc();  
  setTimeout(() => checkRef(weakPersonRef), 1000); // false  
}, 1000);
```

## Proxy

Proxies erlauben, alle Zugriffe auf ein Objekt zu überwachen und bei Bedarf, Standardverhalten zu modifizieren.

```
const standort = {  
  land: "Schottland",  
  ort: "Castle Douglas",  
};  
  
const proxy = new Proxy(standort, {  
  get(target, property) {  
    console.log(`Property holen: ${property}`);  
    return target[property];  
  },  
  // set(target, property, value) { ... },  
  // has(target, property) { ... },  
  // defineProperty(target, property, descriptor) { ... },  
  // deleteProperty(target, property) { ... },  
  // getOwnPropertyDescriptor(target, property) { ... },  
  // ownKeys(target) { ... },  
  // construct(target, argumentsList, newTarget) { ... },  
  // getPrototypeOf(target) { ... },  
  // setPrototypeOf(target, prototype) { ... },  
  // isExtensible(target) { ... },  
  // preventExtensions(target) { ... },  
  // apply(target, thisArg, argumentsList) { ... },  
});
```

```
console.log(proxy.land);
```

```
const proxyMitReflect = new Proxy(  
  {},  
  {  
    get(target, property) {  
      console.log(`Property holen: ${property}`);  
      return Reflect.get(target, property);  
    },  
    set(target, property, value) {  
      console.log(`Property setzen: ${property}`);  
      return Reflect.set(target, property, value);  
    },  
  }  
);
```

```
proxyMitReflect.planet = "Erde";  
proxyMitReflect.stern = "Sonne";
```

```
console.log(proxyMitReflect.planet);  
console.log(proxyMitReflect.stern);
```

## JavaScript-Module

### Module1.mjs

```
const name = "Oliver Sturm";  
const alter = 49;  
  
function getNameUndAlter() {  
  return name + " ist " + alter + " Jahre alt";  
}  
  
export { name, alter, getNameUndAlter };
```

### Module2.mjs

```
const name = "Dr. Holger Schwichtenberg";  
const alter = 50;  
  
function getNameUndAlter() {  
  return name + " ist " + alter + " Jahre alt";  
}  
  
export { name, alter, getNameUndAlter };
```

### ModulNutzer1.mjs: Einfacher Import

```
import { name, alter, getNameUndAlter } from "./Module1.mjs";  
console.log(name + " ist " + alter + " Jahre alt");  
console.log(getNameUndAlter());
```

### ModulNutzer2.mjs: Import mit Aliasen für einzelne Exporte

```
import { name as name2, alter as alter2, getNameUndAlter as  
getNameUndAlter2 } from "./Module2.mjs";  
console.log(name2 + " ist " + alter2 + " Jahre alt");  
console.log(getNameUndAlter2());
```

### ModulNutzer3.mjs: Import mit Alias für das ganze Modul

```
import * as mod2 from "./Module2.mjs";  
console.log(mod2.name + " ist " + mod2.alter + " Jahre alt");  
console.log(mod2.getNameUndAlter());
```

## JSDoc

Kommentare, die nach dem JSDoc-Standard (<https://jsdoc.app>) formatiert sind, werden in der Entwicklungsumgebung als Tooltips angezeigt. Mittels des npm-Pakets "jsdoc" kann automatisch Dokumentation aus den Kommentaren generiert werden. Es gibt auch eslint-Plugins, welche die Kommentare prüfen.

```
/**  
 * Addiert zwei Zahlen. */
```

```
* @param {number} x Der erste Wert  
* @param {number} y Der zweite Wert  
* @returns {number} Die Summe der beiden Werte  
*/  
function add(x, y) {  
  return x + y;  
}
```

```
22  
23  
24  
25  
26  
27  
28  
29  
30  
var x = add(1, 1);
```

```
function add(x: number, y: number): number  
  Addiert zwei Zahlen.  
  @param x — Der erste Wert  
  @param y — Der zweite Wert  
  @returns — Die Summe der beiden Werte
```

So sollte die Funktion aufgerufen werden

```
console.log(add(1, 2)); // 3
```

So sollte die Funktion eigentlich nicht aufgerufen werden.

JSDoc erzwingt allerdings nichts!

```
console.log(add("1", "2")); // "12"
```