

# Modernes JavaScript – ES Versionen, Typen, Scopes, Closures und Funktionen

2021

- Einführung in modernes JavaScript
- Vorstellung der wichtigsten Funktionen, die wir mit React verwenden werden
- Häufige Missverständnisse ausräumen
- JS und DOM-API üben



- Typen und Type System
- Funktionen und Klassen, Scope und Closure
- Template Literale, Shorthand Properties, Objektdestrukturierung, Standards, Optionale Verkettungen
- Array Methoden, Rest/Spread
- ES Module, Web APIs
- Linters und Linting

- ECMAScript -> Sprachspezifikation
- JavaScript:
  - Ursprünglich Mozilla's Implementierung
  - IE hat Jscript-Engine, Chakra
  - Heute bezeichnen alle Browser ihre Implementierung als JavaScript
    - (Aber mit verschiedenen Engines: V8, Gecko, JavaScriptCore, Chakra)



- ECMAScript 3 -> veraltet
- ECMAScript 5 (2009) -> auch veraltet
- **ECMAScript 2015** -> „compilation target“
  - Moderne Features
  - Moderne Engines
  - Unsere Wahl
- ES2016, ES2017...ES2020
- Babel

- Typen
- Zuweisungen und Scope, Closure, Klassen und Funktionen
- „Moderne“ Features
- Web APIs

- JavaScript hat primitive und nicht primitive Typen
- Welche sind welche?



- JS hat primitive Typen: string, number, *bigint*, boolean, undefined, *symbol* und null.
- Alle anderen sind nicht primitive Typen.

- string, number, boolean, null, undefined
- Unveränderbare Werte
- Vergleich “by value”

- Objekte
- globals: String, Number, Boolean, Object, Function, Math, RegExp, window, global ...

# „Everything is an Object“


- Alles ist ein Objekt in JavaScript – Werte, Variablen, Primitive Werte, Objekte, Funktionen, Globals, APIs...

- "str" oder 'ing'
- *„Einfache Anführungszeichen“* bevorzugt
- Wir können diese mit + zusammenführen:

```
"str" + 'ing'
```

- Instanzen haben Methoden: 'str'.toUpperCase()
- Indexiert:

```
'str'[2] === 'r'
```



```
"Zlatko" === 'Zlatko';
```



# Was ist *global*?

- In Browsern: *window* Objekt
- Nicht im main JS Tab: (Web Workers, NodeJS, etc): ???
- [https://developer.mozilla.org/en-US/docs/Glossary/Global\\_object](https://developer.mozilla.org/en-US/docs/Glossary/Global_object)

JavaScript built-in: `globalThis`

Usage: Global 91.04% of all users

Current aligned Usage relative Date relative Filtered All

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Opera Mobile	Chrome for Android	Firefox for Android	UC Browser for Android	Samsung Internet	QQ Browser	Baidu Browser	KaiOS Browser
	12-18	2-64	4-70	3.1-12	10-57	3.2-12.1							4-9.2			
6-10	79-88	65-85	71-88	12.1-13.1	58-72	12.2-13.7		2.1-4.4.4	12-12.1				10.1-12.0			
11	89	86	89	14	73	14.4	all	81	62	88	86	12.12	13.0	10.4	7.12	2.5
		87-88	90-92	TP												

Notes Test on a real browser Feedback

See full reference on [MDN Web Docs](#).

Support data for this feature provided by: MDN browser-compat-data

- global.String (oder window.String in alten Browsern)
- String *Objekt* vs String *Primitive*

```
new String('Zlatko') !== 'Zlatko';
```



```
String('Zlatko') === 'Zlatko'
```

(z.B. bei einer Antwort aus dem Backend)



## *Primitiver Typ* hat alle Methoden des String Objekts

- Alle Methoden des String Konstruktors sind gültig
- `'str'.includes('s')`
- `slice`, `search`, `startsWith`, `substr`, `substring`, `replace`, `trim`...

- `str.charCodeAt`, `str.charAt`, `str.localeCompare`

```
const a = 'réservé'; // with accents,  
lowercase  
const b = 'RESERVE'; // no accents,  
uppercase  
  
console.log(a.localeCompare(b));  
// expected output: 1  
console.log(a.localeCompare(b, 'en', {  
sensitivity: 'base' }));  
// expected output: 0
```

# String Konkatination, Mehrzeilige Strings

```
let greetingMessage = 'Welcome ';  
greetingMessage = user ? greetingMessage + user.name : 'visitor';  
const result = 'Result is ' + 2 + 3 + '.';  
const multiLineString = 'msg systems ag\nRobert-Bürkle-Str. 1\nIsmaning';
```

```
const greetingMessage = `Welcome to ${user ? user.name : 'visitor'}`;
const result = `Result is ${2 + 3}.`;
const multiLineString = `msg systems ag
Robert-Bürkle-Str. 1
Ismaning`;
```

```
const a = 10;  
const b = '30';  
const c = 40;
```

```
const a = 10;  
const b = '30';  
const c = 40;  
return a + b; // returns '1030' (concatenation)  
return b + c; // returns '3040' (concatenation)  
return a + c; // returns 50 (addition)
```

*Template Literale* sind Strings die auch Platzhalterwerte enthalten können.

Es können damit auch einfach mehrzeilige Strings erstellt werden.

Man kann auch „*Tagged Template Literale*“ konstruieren (diese kommen häufig in Komponenten-Bibliotheken vor).

Die Syntax der *Template Literale* dienen als Hilfe zur Fehlervermeidung.



## String

- *"The JavaScript Number type is a double-precision 64-bit binary format IEEE 754 value, like double in Java or C#" – MDN*
- Literalwerte: 1, 2, -3.4, 0.5, .15, -2e3...

```
12 === 12.0
```

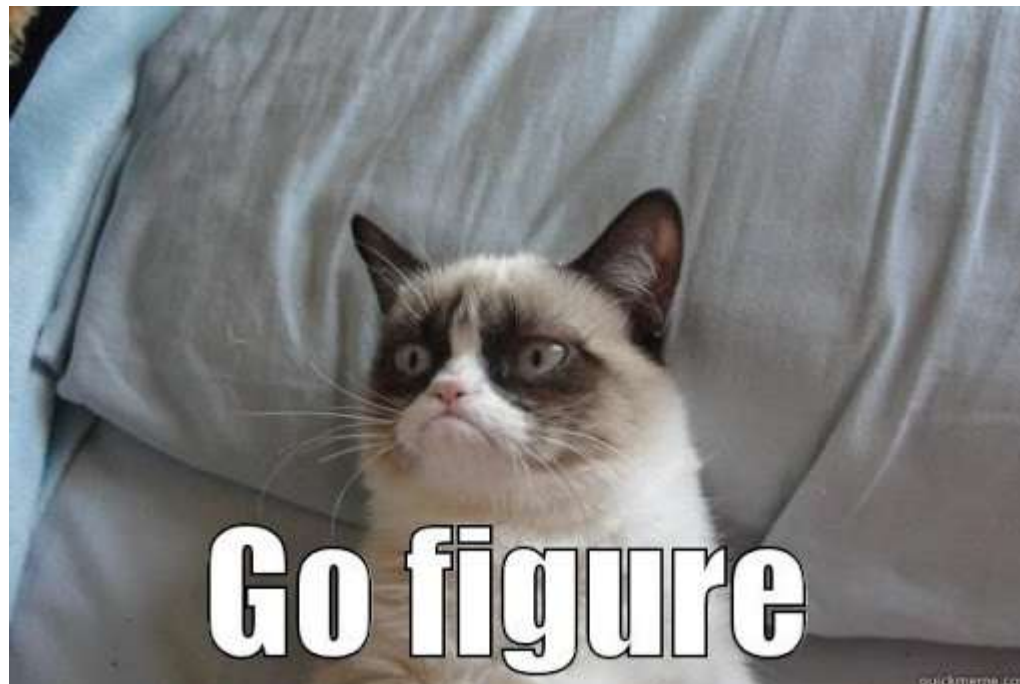
- Nicht JavaScript-spezifisch, FP number

```
.1 + .2 !== .3;
```



- global Number – Wrapper für Zahlen
- global Math – Objekt mit numerischen Konstanten, Methoden,...

```
Number('string or number')  
new Number("2") !== 2  
Number("2") === 2
```



# Wichtige Standards von *Number*

Number.POSITIVE\_INFINITY

Number.NEGATIVE\_INFINITY

Number.MIN\_VALUE

Number.MAX\_VALUE

Number.EPSILON

Number.MAX\_SAFE\_INTEGER →  $(25^3 - 1)$

```
1e3 === 10e2  
10e2 === 100e1  
1e3 === 100e1
```

```
1 * 10^3 === 10 * 10^2  
...
```



# „Not a Number“

*Number*.NaN

```
isNaN(Number("blah"))
```

```
isNaN(2)
```

```
Number.isFinite(3/0)  
Number.isInteger(.15  
Number.isSafeInteger
```

```
Math.sqrt(42)  
Math.acos()  
Math.min()  
Math.max()  
Math.random()
```

- *true* oder *false*
- Primitiver Typ ist kein Objekt => *false !== new Boolean(false)*
- Objekte in Booleans verwandeln: *!!obj*
  
- “Leere” primitive Werte sind false
- “Jedes Objekt, dessen Wert nicht *undefined* oder *null* entspricht, einschließlich eines Boolean-Objektes, dessen Wert *false* ist, wird bei der Übergabe an eine Bedingung als *true* gewertet.” - [MDN](#)

```
const bNoParam = new Boolean();  
const bZero = new Boolean(0);  
const bNull = new Boolean(null);  
const bEmptyString = new Boolean('');  
const bfalse = new Boolean(false);
```

```
const btrue = new Boolean(true);  
const btrueString = new Boolean('tru  
const bfalseString = new Boolean('fa  
const bSuLin = new Boolean('Su Lin')  
const bArrayProto = new Boolean([]);  
const bObjProto = new Boolean({});
```

`"2" == 2` oder `"2" === 2`

`null == false` oder `null === false`



- *“Jedes Objekt, dessen Wert nicht undefined oder null entspricht, einschließlich eines Boolean-Objektes, dessen Wert false ist, wird bei der Übergabe an eine Bedingung als true gewertet.”*

```
if (condition) {  
  ...  
}
```

- `null !== undefined`
- beide sind *falsy*
- *Null* ist ein expliziter Wert
- *Undefined* ist ein fehlender Wert



- *undefined* – nicht definiert

```
var x;  
console.log(x); // -> undefined
```

```
const y = {  
  val: 15,  
}
```

```
y.name // -> undefined;
```

- *null* – expliziter Wert

```
const name = null;  
y.otherValue = null;
```

# Wo ist es wichtig?

```
null === undefined;  
// -> false
```

```
if (condition) {  
  ...  
}
```



`var x = 3;` ← ES5

`const x = 'string';` ← ES2015

`let y = new Number('42');` ← ES2015

Wir nutzen immer *let* und *const*, nicht *var*

# Warum?

- Wir verwenden immer *const*, um Fehler zu vermeiden - versehentliche Neuzuweisungen, Scope-Fehler (Hebefeher) usw.
- Wenn der Wert primitiv ist und geändert werden muss, verwenden wir *let*
- Wenn die Variable neu zugewiesen werden soll, verwenden wir *let*

# Objekt Literale vs Primitive Literale Beispiel

```
const obj = {  
  number: 3,  
  name: string,  
  greet: function(name) {  
    this.name = name || '';  
    alert(`Hello, ${this.name}`);  
  }  
}  
obj.number = 45;
```

```
const x = 6;  
x = 3;  
// Uncaught TypeError: invalid assignment to const 'x'
```

- Objekt *obj* **Reference** ist konstant (aber Objekt selber ist *mutable*)
- Primitive *x* ist *immutable* (unveränderbar)

```
x = 5; // Assign 5 to x

elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element

var x; // Declare x
```

Sieht es richtig aus?



# Was gibt dieser Code aus?

```
console.log('First:', inc(1));  
var increment = 1;  
console.log('Second:', inc(1));  
increment = 3;  
console.log('Third:', inc(1));
```

```
function inc(x) {  
    return x + increment;  
}  
  
console.log('First:',  
inc(1));  
let increment = 1; // oder  
const
```

- *var* Variablen sind „hoisted“ – nach oben gezogen
- Das führt zu unerwarteten Situationen
- Immer *let* und *const* nutzen, da diese nicht „nach oben gezogen“ werden können und „*Block-Scoped*“ sind

- *let* und *const* sind „Block-Scoped“
- *Blöcke* sind mit {} begrenzt

```
function printDrei() {  
  const val = 3;  
  console.log(val); //  
okay  
}  
console.log(val); //  
Error!
```

```
for (const i of [1, 2, 3]) {  
  console.log(i); // okay  
}  
console.log(i); Error
```

- `var x` ist eine Deklaration, da wir nicht den Wert der Variable definieren, sondern ihre Existenz
- `var x = 1` ist sowohl eine Deklaration als auch eine Definition. Es ist jedoch getrennt (x wird am Anfang deklariert - „hoisted“, während die Definition in der angegebenen Zeile erfolgt)

```
const x = 3; // declaration
```

```
let name; // definition - `name` ist  
definiert, aber noch immer undefined.
```

- Funktionen sind ein Grundbaustein in JavaScript
- Eine Funktion hat einen Namen, Argumente und einen Body
  - (Funktionen ohne Namen sind anonyme Funktionen)
- Funktionen haben ihren eigenen Scope

```
function add(a, b) {  
    return a + b;  
}
```

```
this.value = 3;  
document.querySelector('button').onclick = function(ev) {  
    console.log('Event:', ev);  
    console.log(this.value); // undefined  
}
```

- Funktionsdeklaration vs Funktionsausdruck
- Wie bei Variablenzuweisungen werden Funktionsdeklarationen geladen, bevor der Code ausgeführt wird, während Funktionsausdrücke nur geladen werden, wenn der Interpreter diese Codezeile erreicht.
- Das kann darauf Auswirkungen haben, wann z.B. eine IIFE ausgeführt wird
- In React-Anwendungen werden meistens Ausdrücke genutzt

```
const add = function(a, b) { return a + b; }  
doSomethingWith(add);  
objekt.add = add;
```

- Alles innerhalb von () wird sofort ausgewertet
- Wenn der Rückgabewert eine Funktion ist, kann diese gleich ausgeführt werden
- Es wird *Immediately Invoked Function Expression* genannt

```
(function(x) { alert (x) } )("Hallo");
```



- Eine alternative Form des Funktionsausdrucks
- Kürzer, aber hat Einschränkungen
- Sehr häufig benutzt, da es den *Scope nicht ändert!*

```
const add = (x, y) => x + y;
```

```
const subtract(x, y) => {  
  console.log('Subtracting');  
  return x - y;  
}
```

```
const getObject = (name) => ({  
  value: name  
});
```

- Funktionen haben einen eigenen Scope
- Funktionen haben Zugriff auf den umgebenden Scope, aber die Umgebung hat keinen Zugriff auf den Funktions-Scope
- Was passiert hier?

```
function outer() {  
  const value = 3;  
  
  function inner() {  
    const privateValue = 4;  
    console.log(value, privateValue); // 3, 4  
  }  
  
  console.log(value, privateValue);  
}
```

```
class UsersService {  
  getUsers() {  
    fetch('https://jsonplaceholder.typicode.com/todos/1')  
      .then(function (response){  
        return response.json();  
      })  
      .then(function (json) {  
        console.log(json);  
        this.value = json; // what is this?  
      });  
  }  
}
```

```
class UsersService {
  getUsers() {
    fetch('https://jsonplaceholder.typicode.com/todos/1')
      .then(function (response){
        return response.json();
      })
      .then(function (json) {
        console.log(json);
        this.value = json; // what is this?
      });
  }
}

const usersService = new UsersService();
usersService.getUsers();
// Uncaught (in promise) TypeError: this is undefined
```

```
class UsersService {  
  getUsers() {  
  
    fetch('https://jsonplaceholder.typicode.com/todos/1')  
      .then(response => response.json())  
      .then(json => {  
        console.log(json);  
        this.value = json; // not good  
      });  
  }  
}  
  
const usersService = new UsersService();  
usersService.getUsers();  
console.log(usersService.value);
```

```
function outer() {  
  const value = 3;  
  
  function inner() {  
    const privateValue = 4;  
    console.log(value, privateValue); // 3, 4  
  }  
  
  console.log(value, privateValue); // ReferenceError: privateValue is not defined  
}
```

- Ein Dokument mit einem Texteingabefeld und einem Button erstellen
- Eine Funktion erstellen, die einen Wert des Eingabefeldes erfasst und diesen Wert ausgibt, wenn sich ein Wert im Feld befindet. "NO VALUE", wenn kein Wert vorhanden ist.
- Sicherstellen, dass der Wert ausgegeben wird, wenn es eine nicht leere Zeichenfolge ist (z.B. Leerzeichen)

- Vorlagen für Objekte
- Klassen sind Funktionen mit Eigenschaften
  - Sie sind nicht *hoisted*
  - Klassen haben *Konstruktoren*
  - Klassen können *erweitert* werden
  - *Syntaktischer Zucker*
- *class-Ausdruck* und *Klassendeklarationen*



```
class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
}  
  
const meinAuto = new Car('Ioniq', 2019);
```

- muss den genauen Namen „*constructor*“ haben
- wird automatisch ausgeführt, wenn ein neues Objekt erstellt wird
- wird verwendet, um Objekteigenschaften zu initialisieren

```
class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
  
  hupen() {  
    console.log(`%c ${this.name}: T00T T00T!`, 'background: #222; color: #bada55; font-size: 36px')  
  }  
}  
  
const meinAuto = new Car('Ioniq', 2019);  
  
meinAuto.hupen();
```

```
const bmw = new Car('i8', 2021);  
class Car {  
  constructor(name, year) {  
    this.name = name;  
    this.year = year;  
  }  
} // ReferenceError: can't access lexical declaration 'Car' before initialization
```

```
let Rectangle = class {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
};
```

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
}
```

# Anonymous vs. named

```
// unnamed
let Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: ??
```

```
// named
let Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
console.log(Rectangle.name);
// output: ??
```

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
  // Getter  
  get area() {  
    return this.calcArea();  
  }  
  // Method  
  calcArea() {  
    return this.height * this.width;  
  }  
}  
  
const kvadrat = new Rectangle(10, 10);  
  
// kvadrat.area() oder kvadrat.area?
```

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  
  static displayName = "Point";  
  static distance(a, b) {  
    const dx = a.x - b.x;  
    const dy = a.y - b.y;  
  
    return Math.hypot(dx, dy);  
  }  
}
```

```
const p1 = new Point(5, 5);  
const p2 = new Point(10, 10);  
p1.displayName; // ???  
p1.distance(p1, p2); // ??  
Point.displayName; // ??  
Point.distance(p1, p2); // ??
```



```
class Counter extends HTMLElement {  
  clicked() {  
    this.x++;  
    window.requestAnimationFrame(this.render.bind(this));  
  }  
  
  constructor() {  
    super();  
    this.onclick = this.clicked.bind(this);  
    this.x = 0;  
  }  
  
  connectedCallback() { this.render(); }  
  
  render() {  
    this.textContent = this.x.toString();  
  }  
}  
window.customElements.define('num-counter', Counter);
```

## Beispiel mit *public* Variablen



```
class Counter extends HTMLElement {  
  x = 0;  
  
  clicked() {  
    this.x++;  
    window.requestAnimationFrame(this.render.bind(this));  
  }  
  
  constructor() {  
    super();  
    this.onclick = this.clicked.bind(this);  
  }  
  
  connectedCallback() { this.render(); }  
  
  render() {  
    this.textContent = this.x.toString();  
  }  
}  
window.customElements.define('num-counter', Counter);
```

## Beispiel mit *private* Variablen

```
class Rectangle {  
  constructor(height = 0, width) {  
    this.height = height;  
    this.width = width;  
  }  
}
```

```
new Rectangle().x; // => 0
```

```
class Rectangle {  
  #height = 0;  
  #width;  
  constructor(height, width) {  
    this.#height = height;  
    this.#width = width;  
  }  
}
```

```
new Rectangle().x; // => undefined
```

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  speak() {  
    console.log(`${this.name} makes a noise.`);  
  }  
}
```

```
class Dog extends Animal {  
  constructor(name) {  
    super(name); // call the super class constructor and pass in the name parameter  
  }  
  speak() {  
    console.log(`${this.name} barks.`);  
  }  
}
```

```
let d = new Dog('Mitzie');  
d.speak(); // Mitzie barks.
```

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

```
class Dog extends Animal {  
  constructor(name) {  
    super(name); // call the super class constructor and pass in the name parameter  
  }  
}
```

```
let d = new Dog('Mitzie');  
d.speak(); // Mitzie barks.
```

```
let d = new Dog('Mitzie');  
d instanceof Dog; // ??  
d instanceof Animal; // ??
```

- Eine Klasse für ein benutzerdefiniertes Element namens Counter definieren
- Die Klasse soll einen Button zum Erhöhen, einen Button zum Reduzieren einer Zahl und ein Label zum Zählen enthalten
- Drei Instanzen des Elements sollen im DOM platziert werden
- Beispiel:

```
function someFunction() { alert('Hallo!'); }  
const section = document.createElement('section');  
const button = document.createElement('button');  
button.innerHTML = 'Click me';  
button.onclick = someFunction;  
section.appendChild(button);  
document.body.appendChild(section);
```

# Fragen?

