ECMA SCRIPT 6+

BLOCKSCOPEVARIABLEN UND KONSTANTEN

BLOCK SCOPE MIT LET

```
for (let i = 0; i < a.length; i++) {
    let x = a[i];
    ...
}


let callbacks = [];

for (let i = 0; i <= 2; i++) {
    callbacks[i] = () => i * 2;
}

// callbacks[0]() === 0
// callbacks[1]() === 2
// callbacks[2]() === 4
```

BLOCK SCOPES MIT { ... }

```
{
    function foo () { return 1 }          // foo() === 1
    {
        function foo () { return 2 }      // foo() === 2
    }
                                          // foo() === 1
}

ECMAScript 5:
(function () {
    var foo = function () { return 1; }
    foo() === 1;
    (function () {
        var foo = function () { return 2; }
        foo() === 2;
    })();
    foo() === 1;
})();
```

CONST

```
const PI = 3.141593
PI > 3.0
```

```
ECMAScript 5:
Object.defineProperty(typeof global === "object" ? global : window, "PI", {
    value:        3.141593,
    enumerable:   true,
    writable:     false,
    configurable: false
})
PI > 3.0;
```

TEMPLATE LITERALS

- String interpolation
- Custom interpolation
- Raw string access

STRING INTERPOLATION

```
var customer = { name: "Foo" }

var card = { amount: 7, product: "Bar", unitprice: 42 }

var message = `Hello ${customer.name},
want to buy ${card.amount} ${card.product} for
a total of ${card.amount * card.unitprice} bucks?`
```

CUSTOM INTERPOLATION

```
get (`http://example.com/foo?bar=${bar + baz}&quux=${quux}`)
```

THEMENBLOCK FUNKTIONEN

SPREAD- UND RESTOPERATOR

- Defaultparameter in Funktionen
- Funktionsparameter, Rest-Parameter
- Arrow Funktionen vs. Function-Funktionen

DEFAULT PARAMETER, REST-PARAMETER, SPREAD OPERATOR

DEFAULT PARAMETER

```
function f (x, y = 7, z = 42) {
    return x + y + z
}
```

REST PARAMETER

```
function sum(...theArgs) {
  return theArgs.reduce((previous, current) => {
    return previous + current;
  });
}

console.log(sum(1, 2, 3));
// expected output: 6

console.log(sum(1, 2, 3, 4));
// expected output: 10
```

REST PARAMETER

- Rest Parameter bilden ein Array.
- Methoden wie sort, map, forEach oder pop können direkt angewendet werden.
- Das arguments Objekt ist kein echtes Array.
- Das arguments Objekt hat zusätzliche, spezielle Funktionalität (wie die calleeEigenschaft).

SPREAD OPERATOR

```
var params = [ "hello", true, 7 ]
var other = [ 1, 2, ...params ] // [ 1, 2, "hello", true, 7 ]

function f (x, y, ...a) {
    return (x + y) * a.length   // f(1, 2, ...params) === 9
}


var str = "foo"
var chars = [ ...str ]          // [ "f", "o", "o" ]
```

ARROW FUNCTIONS

ARROW FUNCTIONS

- Der Ausdruck einer Pfeilfunktion ist kürzer als ein Funktionsausdruck
- Kein eigenes this, arguments, super, oder new.target.
- Sie können nicht als/in Konstruktoren verwendet werden.

```
(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression
// gleich zu: => { return expression; }

// Klammern sind optional, wenn nur ein Parametername
vorhanden ist:
(singleParam) => { statements }
singleParam => { statements }

// Die Parameterliste für eine parameterlose Funktion muss mit
einem Klammernpaar geschrieben werden
() => { statements }
```

```
// Der Body kann eingeklammert werden, um ein Objektliteral
Ausdruck zurück zu geben:
params => ({foo: bar})

// Rest Parameter und Default Parameter werden unterstützt
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) =>
{
statements }

// Destrukturierung in der Parameterliste ist ebenfalls
unterstützt
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f(); // 6
```

CLASSES

KLASSEN UND VERERBUNG

- Definition von Klassen
- Methoden in Klassen
- constructor-Method
- Vererbung mit extend
- Das super-Keyword

CLASSES

```
ECMAScript 5 – syntactic sugar: reduced | traditional
var Shape = function (id, x, y) {
    this.id = id;
    this.move(x, y);
};
Shape.prototype.move = function (x, y) {
    this.x = x;
    this.y = y;
};
```

CLASSES

NEUE METHODEN FÜR STANDARDOBJEKTE IN ES2015

NEUE METHODEN VON ARRAY

ARRAY METHODS

```
      .isArray()


Array.prototype
      .concat()
      .copyWithin()
      .entries()
      .every()
      .fill()
      .filter()


      .flat()
      .flatMap()
      .forEach()
      .includes()
      .indexOf()
      .join()
      .keys()
      .lastIndexOf()
      .map()
      .pop()
      .push()
      .reduce()
      .reduceRight()
      .reverse()
      .shift()
      .slice()
      .some()
      .sort()
      .splice()
      .toLocaleString()
      .toSource()
      .toString()
      .unshift()
      .values()
```

```
const arrayLike = { length: 2, 0: 'a', 1: 'b' };

// for-of only works with iterable values
for (const x of arrayLike) { // TypeError
    console.log(x);
}

const arr =             (arrayLike);
for (const x of arr) { // OK, iterable
    console.log(x);
}
// Output:
// a
// b
```

```
const spans = document.querySelectorAll('span.name');

// map(), generically:
const names1 = Array.prototype.map.call(spans, s =>
s.textContent);

// Array.from():
const names2 =          (spans, s => s.textContent);
```

```
(item_0, item_1, ···)
```

creates an Array whose elements are item_0, item_1, etc.

```
ECMAScript 5
[ 1, 3, 4, 2 ].filter(
   function (x) {
      return x > 3;
   }
)[0]; // 4
```

TYPED ARRAYS

```
// create a TypedArray with a size in bytes
const typedArray1 =                    ;
typedArray1[0] = 32;

const typedArray2 =                                ;
typedArray2[1] = 42;

console.log(typedArray1);
// expected output: Int8Array [32, 0, 0, 0, 0, 0, 0, 0]

console.log(typedArray2);
// expected output: Int8Array [32, 42, 0, 0, 0, 0, 0, 0]
```

TYPED ARRAYS

```
Int8Array();
Uint8Array();
Uint8ClampedArray();
Int16Array();
Uint16Array();
Int32Array();
Uint32Array();
Float32Array();
Float64Array();
BigInt64Array();
BigUint64Array();
```

NEUE METHODEN VON OBJECT

OBJECT METHODS

```
Object

      .create()
      .defineProperties()
      .defineProperty()
      .entries()
      .freeze()
      .fromEntries()
      .getOwnPropertyDescriptor()
      .getOwnPropertyDescriptors()
      .getOwnPropertyNames()
      .getOwnPropertySymbols()
      .getPrototypeOf()
      .is()
      .isExtensible()
      .isFrozen()
      .isSealed()
      .keys()
      .preventExtensions()
      .seal()
      .setPrototypeOf()
      .values()

Object.prototype
      .__defineGetter__()
      .__defineSetter__()
      __lookupGetter__()
      .__lookupSetter__()
      .hasOwnProperty()
      .isPrototypeOf()
      .propertyIsEnumerable()
      .toLocaleString()
      .toSource()
      .toString()
      .valueOf()
      .watch()
```

ENHANCED OBJECT PROPERTIES

```
ECMAScript 5
var dest = { quux: 0 };
var src1 = { foo: 1, bar: 2 };
var src2 = { foo: 3, baz: 4 };
Object.keys(src1).forEach(function(k) {
    dest[k] = src1[k];
});
Object.keys(src2).forEach(function(k) {
    dest[k] = src2[k];
});
dest.quux === 0; dest.foo  === 3; dest.bar  === 2; dest.baz  === 4;
```

ENHANCED OBJECT PROPERTIES

```
ECMAScript 5
var x = 0, y = 0;
obj = { x: x, y: y };
```

COMPUTED PROPERTY NAMES

```
ECMAScript 5
var obj = {
    foo: "bar"
};
obj[ "baz" + quux() ] = 42;
```

METHOD PROPERTIES

```
ECMAScript 5
obj = {
    foo: function (a, b) {
        …
    },
    bar: function (x, y) {
        …
    },
    //  quux: no equivalent in ES5
    …
};
```

NEUE METHODEN VON STRING, NUMBER ETC.

STRING METHODS

```
String
        .fromCharCode()
        .fromCodePoint()
        .raw()

String.prototype
        .anchor()
        .big()
        .blink()
        .bold()
        .charAt()
        .charCodeAt()
        .codePointAt()
        .concat()

        .fixed()
        .fontcolor()
        .fontsize()

        .indexOf()
        .italics()
        .lastIndexOf()
        .link()
        .localeCompare()
        .match()
        .matchAll()
        .normalize()
        .padEnd()
        .padStart()

        .replace()
        .search()
        .slice()
        .small()
        .split()

        .strike()
        .sub()
        .substr()
        .substring()
        .sup()
        .toLocaleLowerCase()
        .toLocaleUpperCase()
        .toLowerCase()
        .toSource()
        .toString()
```

NEW STRING METHODS

```
"hello"                                  // true
"hello"                                  // true
"hello"                                  // true
"hello"                                  // true
"hello"                                  // false
```

```
ECMAScript 5 "hello".indexOf("ello") === 1;    // true
"hello".indexOf("hell") === (4 - "hell".length); // true
"hello".indexOf("ell") !== -1;    // true
"hello".indexOf("ell", 1) !== -1; // true
"hello".indexOf("ell", 2) !== -1; // false
```

NUMBER TYPE CHECKING

```
Number.            === false
Number.             === true

Number.                  === false
Number.                   === false
Number.               === false
Number.               === true


ECMAScript 5
var isNaN = function (n) {
    return n !== n;
};
var isFinite = function (v) {
    return (typeof v === "number" && !isNaN(v) && v !==
Infinity && v !== -Infinity);
};
```

NUMBER SAFETY CHECKING

```
Number.                          === true
Number.                                  === false
```

```
ECMAScript 5 – syntactic sugar: reduced | traditional
function isSafeInteger (n) {
    return (
            typeof n === 'number'
        && Math.round(n) === n
        && -(Math.pow(2, 53) - 1) <= n
        && n <= (Math.pow(2, 53) - 1)
    );
}
```

STANDARD EPSILON FOR PRECISE FLOATING POINT COMPARISON

```
console.log(0.1 + 0.2 === 0.3) // false

console.log(Math.abs((0.1 + 0.2) - 0.3) <                    ) //
true
```

TRUNC

```
console.log(                    ) // 42
console.log(                    ) // 0
console.log(                    ) // -0
```

```
ECMAScript 5
function mathTrunc (x) {
    return (x < 0 ? Math.ceil(x) : Math.floor(x));
}
```

NUMBER SIGN DETERMINATION

```
console.log(              )   // 1
console.log(              )   // 0
console.log(               )  // -0
console.log(               )  // -1
console.log(                 ) // NaN
```

```
ECMAScript 5
function mathSign (x) {
    return (
        (x === 0 || isNaN(x)) ? x : (x > 0 ? 1 : -1)
    );
}
```

PROMISES

PROMISES

- Bildung und Einsatz von Promises
- Methoden then(), catch()

USING .THEN() ONLY

```
let promise = new Promise(function (resolve, reject) {
  // do a thing, possibly async, then…
  setTimeout(function () {
    try {
      resolve('Promise fullfilled');
    } catch (e) {
      reject(Error("It broke"));
    } finally {
      console.log('Promise ready.')
    }
  }, 2000);
});


promise
    .then(function (result) {
      console.log(result);
    }, function (err) {
      console.log(err);
    });
```

USING .CATCH()

```
let promise = new Promise(function (resolve, reject) {
  // do a thing, possibly async, then…
  setTimeout(function () {
    try {
      resolve('Promise fullfilled');
    } catch (e) {
      reject(Error("It broke"));
    } finally {
      console.log('Promise ready.')
    }
  }, 2000);
});


promise
    .then(function (result) {
      console.log(result);
    })
    .catch(function (err) {
      console.log(err);
    })
   .finally(function () {});

console.log();
```

GENERATOREN

GENERATOREN

- Generatorfunction und Generatorobject
- yield-Keyword und next-Methode

GENERATOR FUNCTION

```
function* range (start, end, step) {
    while (start < end) {
        yield start                                  // yield -> Ertrag
        start += step
    }
}

for (let i of range(0, 10, 2)) {
    console.log(i)                                   // 0, 2, 4, 6, 8
}
```

```
ECMAScript 5
function range (start, end, step) {
    var list = [];
    while (start < end) {
        list.push(start);
        start += step;
    }
    return list;
}

var r = range(0, 10, 2);
for (var i = 0; i < r.length; i++) {
    console.log(r[i]); // 0, 2, 4, 6, 8
}
```

GENERATOR, YIELD UND NEXT()

```
function* foo(index) {
  while (index < 2) {
    yield index++;
  }
}

const iterator = foo(0);

console.log(iterator.next().value);   // expected output: 0
console.log(iterator.next().value);   // expected output: 1
```

GENERATOR FUNCTION MIT ITERATOR

```
let fibonacci = {
  *[Symbol.iterator]() {
    let previous = 0, current = 1
    for (;;) {
      [ previous, current ] = [ current, previous + current ];
      yield current;
    }
  }
}

for (let n of fibonacci) {
    if (n > 1000)
        break
    console.log(n)
}
```

SYMBOLS

SYMBOL

```
const symbol1 = Symbol();
const symbol2 = Symbol(42);
const symbol3 = Symbol('foo');

console.log(typeof symbol1);
// expected output: "symbol"

console.log(symbol3.toString());
// expected output: "Symbol(foo)"

console.log(Symbol('foo') === Symbol('foo'));
// expected output: false
```

SYMBOL()

- The Symbol() function returns a value of type symbol
- Static properties that expose several members of built-in objects
- Static methods that expose the global symbol registry
- Resembles a built-in object class but is incomplete as a constructor because it does not support the syntax "new Symbol()"
- Every symbol value returned from Symbol() is unique.

SYMBOL

Properties

```
Symbol
   .asyncIterator
   .hasInstance
   .isConcatSpreadable
   .iterator
   .match
   .matchAll
   .prototype
   .prototype.description
   .replace
   .search
   .species
   .split
   .toPrimitive
   .toStringTag
   .unscopables
```

Methods

```
Symbol
   .for()
   .keyFor()

Symbol.prototype
   .toSource()
   .toString()
   .valueOf()
```

[SYMBOL.ITERATOR]

```
const iterable1 = new Object();

iterable1[Symbol.iterator] = function* () {
  yield 1;
  yield 2;
  yield 3;
};

console.log([...iterable1]); // expected output: Array [1, 2, 3]
```

ITERATOREN

FOR ... OF

```
let fibonacci = {
    [Symbol.iterator]() {
        let pre = 0, cur = 1
        return {
           next () {
               [ pre, cur ] = [ cur, pre + cur ]
               return { done: false, value: cur }
           }
        }
    }
}

for (let n of fibonacci) {
    if (n > 1000)
        break
    console.log(n)
}
```

SPREAD- UND RESTOPERATOR

- for-of-Schleife
- Bildung und Einsatz von Iteratoren

MAP/SET,
WEAKMAP/WEAKSET

SET()

```
let s = new Set();
s.add("hello").add("goodbye").add("hello");

                                        // s.size === 2
                                        // s.has("hello") ===
true

for (let key of s.values()) {           // insertion order!!!
    console.log(key);
}
```

MAP()

```
let m = new Map();
let s = Symbol();

m.set("hello", 42);                 // m.get(s) === 34
m.set(s, 34)                        // m.size === 2

for (let [ key, val ] of m.entries()) {
    console.log(key + " = " + val);
}
```

WEAKSET, WEAKMAP()

```
let isMarked = new WeakSet();

export class Node {
    constructor (id)   { this.id = id }
    mark         ()    { isMarked.add(this) }
    unmark       ()    { isMarked.delete(this) }
    marked       ()    { return isMarked.has(this) }
}

let foo = new Node("foo") // JSON.stringify(foo) === '{"id":"foo"}'

foo.mark()                // JSON.stringify(foo) === '{"id":"foo"}'
                          // isMarked.has(foo)     === true

foo = null                // remove only reference to foo
                          // isMarked.has(foo)     === false
```

WEAKSET, WEAKMAP()

```
let attachedData = new WeakMap()

export class Node {
    constructor (id)   { this.id = id                    }
    set data    (data) { attachedData.set(this, data)  }
    get data    ()     { return attachedData.get(this) }
}

let foo = new Node("foo")     // JSON.stringify(foo) === '{"id":"foo"}'

foo.data = "bar"              // foo.data === "bar"
                              // JSON.stringify(foo) === '{"id":"foo"}'
                              // attachedData.has(foo) === true

foo = null                    // remove only reference to foo
                              // attachedData.has(foo) === false
```

DESTRUCTURING VON ARRAYS UND OBJEKTEN

DESTRUCTURING

- Die destrukturierende Zuweisung ermöglicht es, Daten aus Arrays oder Objekten zu extrahieren

- Die Syntax ist der Konstruktion von Array- und Objekt-Literalen nachempfunden.

- Destructuring ist "fail-soft", ähnlich wie Standardobjekte, die nach `foo["bar"]`, schauen, und ggf. nur ein `undefined` liefern.

OBJECT AND ARRAY MATCHING

```
var a, b, rest;
[a, b] = [10, 20];
```

```
// a === 10
// b === 20
```

```
[a, b, ...rest] = [10, 20, 30, 40, 50];
```

```
// a === 10
// b === 20
// rest === [30, 40, 50]
```

```
{ a, b } = { c: 10, d: 20 };
```

```
// a === 10
// b === 20
```

OBJECT AND ARRAY MATCHING, DEFAULT VALUES

```
// Fail-soft destructuring
var [a] = [];                    // a === undefined;


// Fail-soft destructuring with defaults
var [a = 1] = [];                // a === 1;
```

SPREAD- UND RESTOPERATOR

- Destructuring von Arrays
- Destructuring von Objekten

REFLECTIONS

REFLECT

```
let obj = { a: 1 }

Object.defineProperty(obj, "b", { value: 2 })
obj[Symbol("c")] = 3

console.log(Reflect.ownKeys(obj))          // [ "a", "b", Symbol(c) ]
```

REFLECT

Reflect.apply()
Ruft eine Zielfunktion mit Argumenten auf, die Argumente werden im Parameter args angegeben. Siehe auch Function.prototype.apply().

Reflect.construct()
Der new operator als Funktion. Equivalent zu new target(...args). Bietet die optionale Möglichkeit, einen anderen Prototyp anzugeben.

Reflect.defineProperty()
Ähnlich zu Object.defineProperty(). Gibt einen Boolean zurück.

Reflect.deleteProperty()
Der delete operator als Funktion. Ähnlich zu dem Aufruf delete target[name].

Reflect.get()
Eine Funktion, die den Wert von Eigenschaften/Properties zurückgibt.

Reflect.getOwnPropertyDescriptor()
Ähnlich zu Object.getOwnPropertyDescriptor(). Gibt einen Eigenschaftsdeskriptor der angegebenen Eigenschaft, oder undefined zurück.

Reflect.getPrototypeOf()
Gleich wie Object.getPrototypeOf().

REFLECT

Reflect.has()
Der in operator als Funktion. Gibt einen booleschen Wert zurück, der angibt, ob eine eigene oder geerbte Eigenschaft vorhanden ist.

Reflect.isExtensible()
Gleich wie Object.isExtensible().

Reflect.ownKeys()
Gibt ein Array der eigenen (nicht geerbten) Eigenschaftsschlüssel des Zielobjekts zurück.

Reflect.preventExtensions()
Ähnlich zu Object.preventExtensions(). Gibt einen Boolean zurück.

Reflect.set()
Eine Funktion, die den Eigenschaften/Properties Werte zuweist. Gibt einen Booleanzurück, der true ist, wenn die Zuweisung erfolgreich verlief.

Reflect.setPrototypeOf()
Eine Funktion, die den Prototyp eines Objekts festlegt.

PROXIES

PROXY

```
let target = {
  foo: "Welcome, foo"
}

let proxy = new Proxy(target, {
  get(receiver, name) {

    if (name in receiver) {
      value = receiver[name];
    } else {
      value = `Hello, ${name}`;
    }

    return value;
  }
})

// proxy.foo === "Welcome, foo"
// proxy.world === "Hello, world"
```

MODULE

MODULE

- Language-level support for modules for component definition.
- Codifies patterns from popular JavaScript module loaders (AMD, CommonJS).
- Runtime behaviour defined by a host-defined default loader.
- Implicitly async model — no code executes until requested modules are available and processed.

MODULES

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js
import * as math from "lib/math";
alert("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```