

# ECMA SCRIPT 6+

# BLOCKSCOPE VARIABLEN 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;
```

# THEMENBLOCK FUNKTIONEN

# SPREAD- UND RESTOPERATOR

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

# DEFAULT PARAMETER, REST- PARAMETER, SPREAD OPERATOR



# DEFAULT PARAMETER

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

```
function f(x){  
    let x = x || 42;  
}
```

```
f() // x=undefined, y=7; z=42
```

# 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

$( ) \Rightarrow \{ \}$

# 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) => expression`

`// gleich zu: => { return expression; }`



```
(singleParam) => { statements }  
(oneParam, twoParam, ..., paramN) => { statements }
```

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

Das ist also auch möglich:  
**`singleParam => returnStatement`**

// Die Parameterliste für eine parameterlose Funktion  
muss mit einem Klammernpaar geschrieben werden:

```
() => { statements }
```

Der Body kann eingeklammert werden, um ein Objektliteral zurück zu geben:

```
params => ({foo: bar})
```

Rest-Parameter und Default-Parameter:

```
(oneParam, oneParam, ...rest) => { statements }
```

```
(param1 = defaultValue) => { statements }
```

# CLASSES

# CLASSES

```
class MyClass {  
    constructor () {}  
  
    getProperty() {}  
    setProperty() {}  
}
```

# CLASSES

```
class ChildClass extends parentClass {  
    constructor () {}  
  
    getProperty() {}  
    setProperty() {}  
}
```

# KLASSEN, KONSTRUKTOR,

```
class Shape {  
  
    constructor (id, x, y) {  
        this.id = id. // property!  
        this.move(x, y)  
    }  
    move (x, y) {  
        this.x = x  
        this.y = y  
    }  
    getId () {}  
    setMove(){}  
}  
  
let myShape = new Shape('rect', 10, 10);
```

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;  
};
```

# VERERBUNG, ELTERNKONSTRUKTOR

```
class Rectangle extends Shape {  
    constructor (id, x, y, width, height) {  
        super(id, x, y);  
        this.width = width;  
        this.height = height;  
    }  
}  
class Circle extends Shape {  
    constructor (id, x, y, radius) {  
        super(id, x, y);  
        this.radius = radius;  
    }  
}
```







# 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}` )
```

# 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

```
// Modul: 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));
```



# VERSCHIEDENE EXPORT MÖGLICHKEITEN

```
export { name1, name2, ..., nameN };  
export function FunctionName(){...}
```

```
export class ClassName {...}  
export default class ClassName {...}
```

```
...
```

```
export * from ...;  
export { name1, name2, ..., nameN } from ...;  
export { import1 as name1, import2 as name2, ..., nameN } from  
...;  
export { default } from ...;
```

# EXPORT EINER KLASSE

```
class User {  
  constructor(n, p, e) {  
    this._name = n;  
    this._prename = p;  
    this._email = e;  
  }  
  
  move() {  
    console.log('1 step');  
  }  
  
  set name(n = 'no name') { this._name = n; }  
  set prename(p) { this._prename = p; }  
  set email(e) { this._email = e; }  
  
  get name() { return this._name; }  
  get prename() { return this._prename; }  
  get email() { return this._email; }  
}
```

```
export default User;
```

# EXPORT EINER KLASSE

```
export class Male extends User {  
  constructor(n, p, e) {  
    super(n, p, e);  
  }  
  drink() {  
    console.log('beer');  
  }  
}  
  
export class Female extends User {  
  constructor(n, p, e) {  
    super(n, p, e);  
  }  
  dance() {  
    console.log('jive');  
  }  
  move() {  
    console.log('2 steps');  
  }  
}
```

# IMPORT VON KLASSE

```
import User from './components/user.js';  
import Male from './components/user.js';  
import Female from './components/user.js';  
  
let admin = new User('Joe');  
log(admin.getName());  
  
let ann = new Female('Doe', 'Ann', 'ann@doe.org');  
log(ann.getPrenome());  
  
let john = new Male('Doe', 'John', 'john@doe.org');  
log(john.getPrenome());
```

# EINBINDEN EINES MODULS

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>ES 6+ Modules</title>
</head>

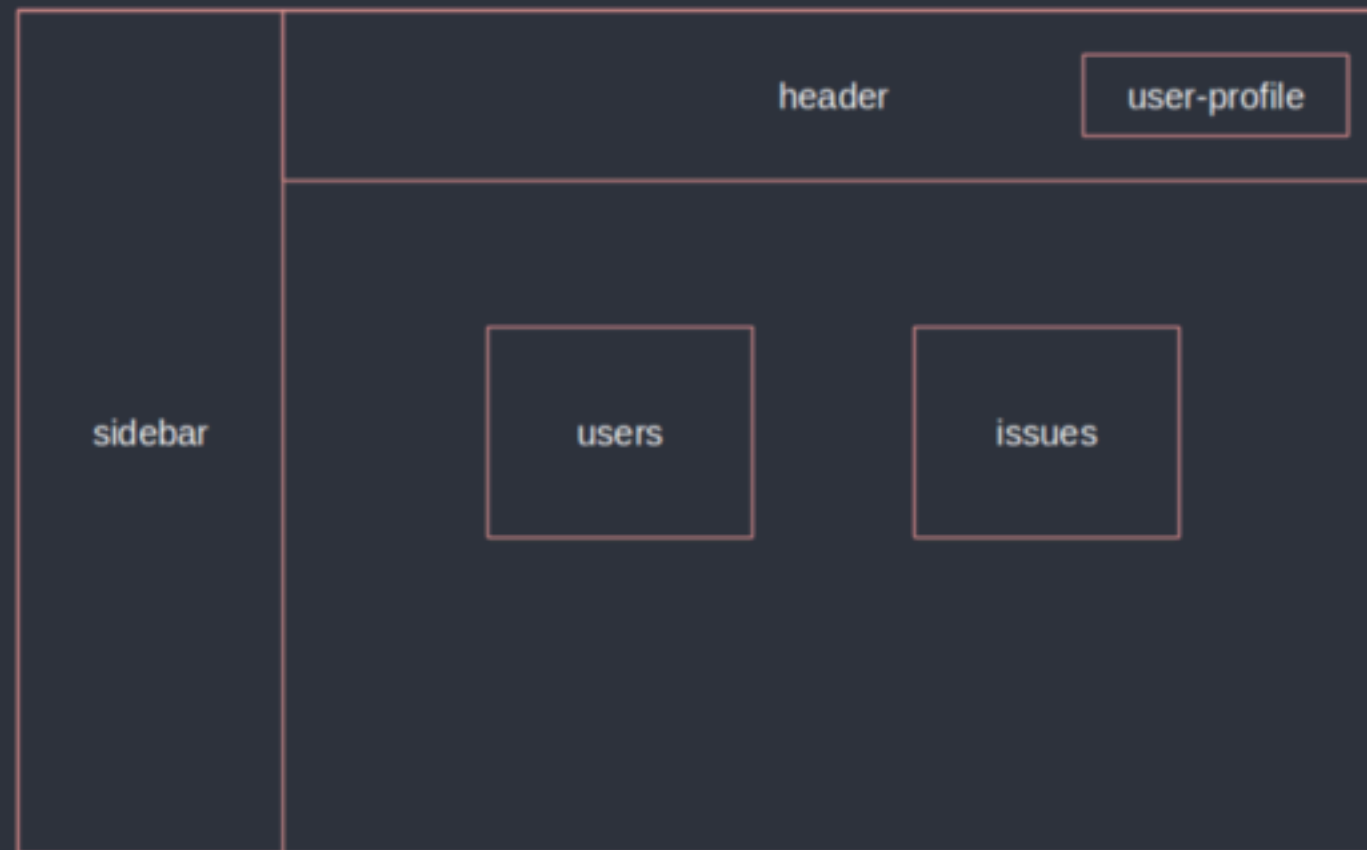
<body>
  <nav>
  </nav>
  <script type="module" src="modules.js"></script>
</body>

</html>
```

# EINE APPLIKATION MIT MODULEN ENTWERFEN



## Dashboard





# SCHRITT 1 - ENTWURF

Ein guter Entwurf erspart Zeit und Schmerzen.  
Ein Entwurf muss nicht perfekt sein, er sollte aber die Richtung weisen.

Entwurf für eine Softwarearchitektur:

**Components:** `users.js`, `user-profile.js`, `issues.js`

**Layouts:** `header.js`, `sidebar.js`

**Dashboard:** `dashboard.js`

Alle Komponenten werden in `dashboard.js` geladen. Das Dashboard wird dann über `index.js` initiiert.



# SCHRITT 2 - ORDNERSTRUKTUR

```
root
|- dashboard
|   |- dashboard.js
|
|- components
|   |- issues.js
|   |- user.js
|   |- userprofile.js
|
|- layouts
|   |- header.js
|   |- sidebar.js
|
|- snippets
|   |- user-data.html | .jade | .ejs
|- index.html
|- index.js ( entry point )
```

# SCHRITT 3 – IMPLEMENTATION

Komponenten bauen: Jede Komponente ist eine Klasse  
Eine Methode zeigt das Laden einer Komponente an.

```
class Users {  
  loadUsers() {  
    console.log('Users component is loaded...')  
  }  
  buildHtml(){  
    // load Data  
    // load template  
    // merge both and return  
  }  
}  
export { Users };
```

# SCHRITT 3 – IMPLEMENTATION

```
import { UserProfile } from '../components/users-profile.js';

class Header {
  loadHeader() {
    // Create a new instance
    const userProfile = new UserProfile();

    // Invoke the method (component)
    userProfile.loadUserProfile();

    // Output loading status
    console.log('Header component is loaded...')
  }
}

export { Header };
```

# SCHRITT 3 – IMPLEMENTATION

```
// From component folder
import { Users } from '../components/users.js';
import { Issues } from '../components/issues.js';

// From layout folder
import { Header } from '../layouts/header.js';
import { Sidebar } from '../layouts/sidebar.js';

class Dashboard {
  loadDashboard(){

    // Create new instances
    const users = new Users();
    const issues = new Issues();
    const header = new Header();
    const sidebar = new Sidebar();

    function addToDashboard(users.buildHtml(), '#user-layout-id'){ ... }

    console.log('Dashboard component is loaded');
  }
}

export { Dashboard }
```

# SCHRITT 3 – IMPLEMENTATION

```
// index.js:  
  
import { Dashboard } from './dashboard/dashboard.js';  
  
const dashboard = new Dashboard();  
dashboard.loadDashboard();
```

<https://www.freecodecamp.org/news/how-to-use-es6-modules-and-why-theyre-important-a9b20b480773/>

# PROMISES

# FETCH() (PROMISES ANWENDEN)

```
fetch('data.json')
  .then(function (response) {
    if (response.ok)
      return response.json();
    else
      throw new Error('Daten konnten nicht geladen werden');
  })
  .then(function (json) {
    // Hier Code zum Abarbeiten der Daten
  })
  .catch(function (err) {
    // Hier Fehlerbehandlung
  });

// fetch basiert auf Promises
```



# PROMISES VERSTEHEN

Stell dir vor, du bist ein Kind. Deine Mutter verspricht dir, dass sie dir nächste Woche ein neues Telefon schenkt.

Du weißt jetzt nicht genau, ob du das Telefon bis nächste Woche wirklich bekommst. Deine Mutter kann es dir schenken, sie kann es aber auch lassen, wenn sie mit Dir nicht zufrieden ist.

Das ist ein **promise**.

Ein **promise** hat 3 Zustände:

**Pending:** Du weißt nicht, ob du das Handy bekommst.

**Fulfilled:** Deine Mutter ist zufrieden, sie kauft und bringt dir das neue Handy.

**Rejected:** Deine Mutter ist nicht zufrieden und besorgt dir kein neues Handy.

# CREATING A PROMISE (ES6)

```
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
  (resolve, reject) => {
    if (isMomHappy) {
      const phone = {
        brand: 'Samsung',
        color: 'black'
      };
      resolve(phone);
    } else {
      const reason = new Error('mom is not happy');
      reject(reason);
    }
  }
);
```

# CONSUMING A PROMISE

```
// call our promise

const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled))
        .catch(error => console.log(error.message));
};

askMom();
```

# CHAINING PROMISES

- Promises are chainable.
- Let's say, you, the kid, promises your friend that you will show them the new phone when your mom buy you one.

# CHAINING PROMISES

```
// 2nd promise
```

```
const showOff = function (phone) {  
  const message = 'Hey friend, I have a new ' +  
    phone.color + ' ' + phone.brand + ' phone';  
  return Promise.resolve(message);  
};
```

# CALL YOUR PROMISES

```
var askMom = function () {  
  willIGetNewPhone  
    .then(showOff) // chain it here  
    .then(function (fulfilled) {  
      console.log(fulfilled);  
      // output: 'Hey friend, I have a new black Samsung  
phone.'  
    })  
    .catch(function (error) {  
      // oops, mom don't buy it  
      console.log(error.message);  
      // output: 'mom is not happy'  
    });  
};
```

# CREATING A PROMISE (ES7)

```
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
  (resolve, reject) => {
    if (isMomHappy) {
      const phone = {
        brand: 'Samsung',
        color: 'black'
      };
      resolve(phone);
    } else {
      const reason = new Error('mom is not happy');
      reject(reason);
    }
  }
);
```

# CHAINING PROMISES

```
// 2nd promise
```

```
async function showOff(phone) {  
  return new Promise(  
    (resolve, reject) => {  
      var message = 'Hey friend, I have a new ' +  
        phone.color + ' ' + phone.brand + ' phone';  
  
      resolve(message);  
    }  
  );  
};
```



# CALL YOUR PROMISES

```
async function askMom() {  
  try {  
    console.log('before asking Mom');  
  
    let phone = await willIGetNewPhone;  
    let message = await showOff(phone);  
  
    console.log(message);  
    console.log('after asking mom');  
  }  
  catch (error) {  
    console.log(error.message);  
  }  
}
```

<https://scotch.io/tutorials/javascript-promises-for-dummies>

# ES7: ASYNC/AWAIT

```
function scaryClown() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('🤡');  
    }, 2000);  
  });  
}
```

```
async function msg() {  
  const msg = await scaryClown();  
  console.log('Message:', msg);  
}
```

```
msg(); // Message: 🤡 <-- after 2 seconds
```

# ES7: ASYNC/AWAIT

```
function who() {  
  return new Promise(resolve =>  
  {  
    setTimeout(() => {  
      resolve('😄');  
    }, 200);  
  });  
}
```

```
function what() {  
  return new Promise(resolve =>  
  {  
    setTimeout(() => {  
      resolve('lurks');  
    }, 300);  
  });  
}
```

```
function where() {  
  return new Promise(resolve =>  
  {  
    setTimeout(() => {  
      resolve('in the shadows');  
    }, 500);  
  });  
}
```

```
async function msg() {  
  const a = await who();  
  const b = await what();  
  const c = await where();  
  
  console.log(`${a} ${b} ${c}`);  
}
```

```
msg(); // 😄 lurks in the  
shadows <-- after 1 second
```

# ES7: ASYNC/AWAIT

```
async function msg() {  
  const a = await who();  
  const b = await what();  
  const c = await where();  
  
  console.log(`${a} ${b} ${c}`);  
}
```

```
async function msg() {  
  const [a, b, c] = await  
    Promise.all([who(), what(), where()]);  
  
  console.log(`${a} ${b} ${c}`);  
}  
  
msg(); // 😊 lurks in the shadows <--  
after 1 second
```

<https://alligator.io/js/async-functions/>

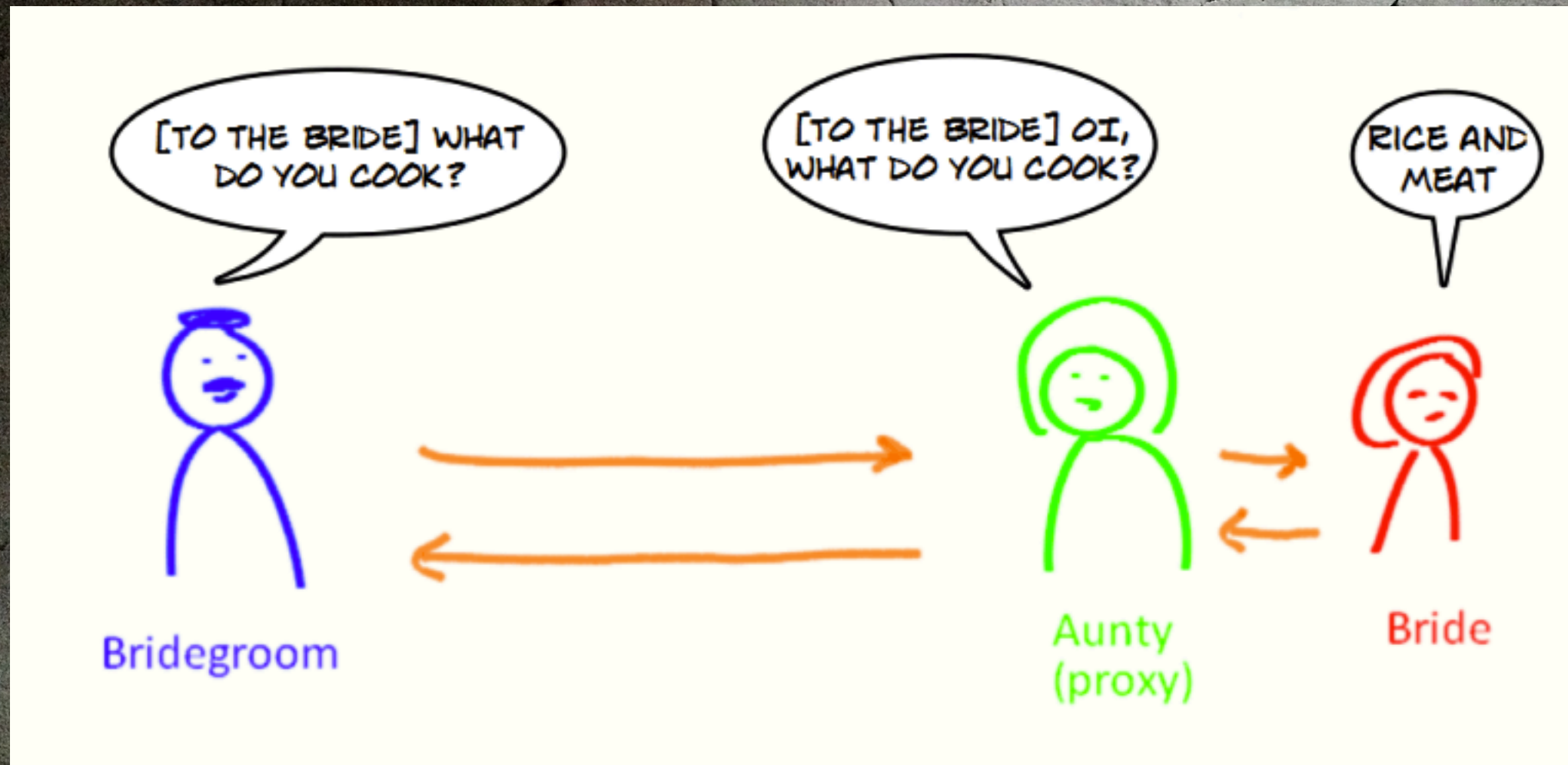






# PROXIES





QUELLE: [HTTPS://DZONE.COM/ARTICLES/SCALA-PROXY-DESIGN-PATTERN](https://dzone.com/articles/scala-proxy-design-pattern)



# PROXY (GRUNDPRINZIP) (STELLVERTRETER, VERMITTLER)

```
let target = {  
  x: 10,  
  y: 20  
}
```

Original

```
let handler = {  
  get: (obj, prop) => 42  
}
```

offiziell: trap (engl.: Falle)

Falle stellen, besser:  
Stellvertreter einrichten

```
target = new Proxy(target, handler)
```

```
target.x // 42  
target.y // 42
```

Ab jetzt übernimmt  
der Stellvertreter das Ergebnis!

# PROXY TRAPS

```
handler.get  
handler.set  
handler.has  
handler.apply  
handler.construct  
handler.ownKeys  
handler.deleteProperty  
handler.defineProperty  
handler.isExtensible  
handler.preventExtensions  
handler.getPrototypeOf  
handler.setPrototypeOf  
handler.getOwnPropertyDescriptor
```



Setze das nachfolgende Beispiel selbstständig um.

20 MINUTEN

# "PRIVATE" - VERSTECKEN VON EIGENSCHAFTEN MIT PROXIES

```
const hide = (target, prefix = '_') => new Proxy(target, {  
  has: (obj, prop) => ( !prop.startsWith(prefix) && prop in obj ),  
  ownKeys: (obj) => Reflect.ownKeys(obj)  
    .filter(prop => (typeof prop !== "string" || !prop.startsWith(prefix))),  
  get: (obj, prop, rec) => (prop in rec) ? obj[prop] : undefined  
})
```

# "PRIVATE" - VERSTECKEN VON EIGENSCHAFTEN MIT PROXIES

```
let userData = hide({  
  firstName: 'John',  
  mediumHandle: '@jdoe',  
  _favoriteRapper: 'Ann'  
})
```

```
console.log( userData._favoriteRapper )  
console.log(('_favoriteRapper' in userData) )  
console.log( Object.keys(userData) )
```

```
// undefined  
// false  
// ['firstName', 'mediumHandle']
```

<https://blog.bitsrc.io/a-practical-guide-to-es6-proxy-229079c3c2f0>

THOMAS BARRASSO

# 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.

# DESTRUCTURING WITH OBJECTS

```
// old way
let tuple = {
  a: 3,
  b: 7,
  c: 6
};
let a = tuple.a;
let b = tuple.b;
let c = tuple.c;

log(a, b, c);

// Destructuring

let { a, b, c } = tuple;
log(a, b, c);
```

# DESTRUCTURING

```
const student = {  
  firstname: 'Ann',  
  lastname: 'Doe',  
  country: 'UK'  
};
```

```
({ firstname, lastname, country } = student); // Ann, Doe, UK
```

```
({ firstname, lastname } = student); // Ann, Doe
```

# USING DIFFERENT NAMES AND DEFAULT VALUES

```
const person = {  
  name: 'John Doe',  
  country: 'US'  
};
```

```
const {  
  name: fullname,      <- setting an alias  
  country: place,  
  age: years = 25      <- a default value  
} = person;
```

```
log(`I am ${fullname} from ${place} and I am ${years} years old.`);  
-> I am John Doe from US and I am 25 years old.
```



Destrukturiere.

```
{prename:"john", name:"doe", email:"john@doe.org"}
```

```
console.log(`Hello, my name is ${fullname}`);  
console.log(`You can send me an Email to ${email}`);  
console.log(`The best time would be ${bestTime}`);
```

10 MINUTEN

# DESTRUCTURING WITH ARRAYS

```
const rgb = [255, 200, 0];  
const [red, green, blue] = rgb;  
  
log(`R: ${red}, G: ${green}, B: ${blue}`);  
R: 255, G: 200, B: 0
```

# SKIPPING ITEMS

```
const rgba = [200, 255, 100, 1];
```

```
// Skip the first two items
```

```
const [ , , blue] = rgba;
```

```
log(`Blue: ${blue}`);
```

```
R: 200, G: undefined, B: 255
```

# USING THE SPREAD OPERATOR

```
const rainbow = ['red', 'orange', 'yellow', 'green', 'blue',  
                'indigo', 'violet'];
```

```
// Assign the first and third items to red and yellow  
// Assign the remaining items to otherColors using the spread operator (...)
```

```
const [red, , yellow, ...otherColors] = rainbow;
```

```
log(otherColors);  
green, blue, indigo, violet
```



# USING THE SPREAD OPERATOR AS FUNCTION ARGUMENT

```
function fn(a, b, c, d, ...args) {  
  console.log('rest param:', a, b, args);  
  console.log('number of expected args:', fn.length);  
  console.log('number of additional args:', args.length);  
}
```

```
fn(1, 2, 3, 4, 5, 6, 7, 8, 9, 0);
```

```
rest param:    1    2    5,6,7,8,9,0  
number of expected args:    4  
number of additional args: 6
```



Schreibe eine Funktion, die beliebig viele Zahlen addiert, die über ein Standard- und Rest-Argument übergeben werden.

```
function add(a, ...rest){  
    ...  
    return result; // console.log(result)  
}
```

12 MINUTEN

<https://codeburst.io/es6-destructuring-the-complete-guide-7f842d08b98f>

GLAD CHINDA







# REFLECTIONS

# WAS IST REFLECTION?

- Reflexion die Fähigkeit eines Programms, Variablen, Eigenschaften und Methoden von Objekten zur Laufzeit zu manipulieren.
- ES5 besitzt bereits Reflexionsfunktionen, wenn sie auch nicht offiziell so genannt werden. `Object.keys()`, `Object.getOwnPropertyDescriptor()` und `Array.isArray()` sind klassische Reflections.

# REFLECT()

- ES6 führt ein neues globales Objekt namens Reflect ein.
- Es ermöglicht,
  - > Methoden aufzurufen,
  - > Objekte zu konstruieren,
  - > Eigenschaften abzurufen und zu setzen
  - > Eigenschaften zu manipulieren und zu erweitern.

# 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.

# EIN OBJEKT ERZEUGEN: REFLECT.CONSTRUCT()

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }
};

let args = ['John', 'Doe'];

let john = Reflect.construct(      // -> john = new Person()
  Person,
  args
);

console.log(john instanceof Person); // -> true
console.log(john.fullName);          // -> John Doe
```

# EINE FUNKTION AUFRUFEN: REFLECT.APPLY()

```
// Old way
let result = Function.prototype.apply.call(Math.max, Math, [10, 20, 30]);

// Reflect.apply(target, thisArg, args)
// New way
let result = Reflect.apply(Math.max, Math, [10, 20, 30]);

console.log(result);
```

# EINE PROPERTY DEFINIEREN: REFLECT.DEFINEPROPERTY()

```
// Reflect.defineProperty(target, propertyName, propertyDescriptor)
```

```
let person = {  
  name: 'John Doe'  
};
```

```
Reflect.defineProperty(person, 'age', {  
  writable: true,  
  configurable: true,  
  enumerable: false,  
  value: 25,  
})) {
```

```
console.log(person.age);
```

<https://www.javascripttutorial.net/es6/javascript-reflection/>

JAVASCRIPT TUTORIAL WEBSITE

# SET, WEAKSET

- ES6 besitzt einen neuen Typ namens `Set` ist ein neuer Typ mehrwertiger Variablen.
- `Set` bildet eine Liste von *eindeutigen* Werten jedes beliebigen Typs.



# SET()

```
let setObject = new Set();
```

```
let chars = new Set(['a', 'b', 'c']);  
// -> Set { 'a', 'b', 'c' }
```

```
let chars = new Set(['a', 'b', 'b', 'c', 'c', 'c']);  
// -> Set { 'a', 'b', 'c' }
```

# SET()

```
let size = chars.size;  
console.log(size);      // 3  
  
chars.add('d').add('e');  
console.log(chars);      // Set {'a','b','c','d',e}  
  
exist = chars.has('z');  
console.log(exist);      // false  
  
chars.delete('e');  
console.log(chars);      // Set {'a','b','c','d'}  
  
chars.clear();  
console.log(chars);      // Set{}
```

# WEAKSET

```
let roles = new Set();

roles.add('admin')
    .add('editor')
    .add('subscriber');

for (let role of roles) {
    console.log(role);
}

for (let [key, value] of roles.entries()) {
    console.log(key, value);
}
```

- Ein WeakSet ist ein Set, es enthält aber nur Objekte.
- Es hat keine Size-Eigenschaft.
- Ein WeakSet ist nicht iterabel. Man verwendet ein WeakSet nur, um zu prüfen, ob ein bestimmter Wert im Set enthalten ist.

```
let computer = {type: 'laptop'};
let server = {type: 'server'};

let equipment = new WeakSet([computer, server]);

if (equipment.has(server)) {
  console.log('We have a server');
}
```

<https://www.javascripttutorial.net/es6/javascript-set/>

# MAP, WEAKMAP

- Schlüssel-Wert-Paare werden bisher mit Objekten abgebildet. Das hat auch einige Nachteile.
- Ein Objekt hat immer einen Standardschlüssel wie **prototype**.
- Ein Schlüssel eines Objekts **muss** eine Zeichenfolge oder ein Symbol sein.
- Ein Objekt hat keine **size**-Eigenschaft.



- Eine Map object beinhaltet key-value Paare.
- Schlüssel und Werte können jeden Typ haben
- Das Map object besitzt eine Reihenfolge. (Die des Einfügens).

# 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 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
```

<https://www.javascripttutorial.net/es6/javascript-map/>

# SYMBOLS

- Neuer Datentyp:  
`Symbol()` erzeugt einen Wert vom Typ `symbol`
- Symbols sind immutable und immer **unique**
- Symbols können nicht implizit zu einem String konvertieren; deshalb `symbol.toString()`

# 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

## 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]()



# FOR ... OF - ARRAY ITERATION

```
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)  
}
```

<https://codeburst.io/a-practical-guide-to-es6-symbol-3fc90117c7ac>

# 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)
}
```





Schreibe einen Generator, der alle natürlichen Zahlen ausgibt.

```
function* naturalNumbers() { ... }  
  
console.log(numbers.next().value)  
console.log(numbers.next().value)  
console.log(numbers.next().value)  
console.log(numbers.next().value)
```

10 MINUTEN

<https://codeburst.io/understanding-generators-in-es6-javascript-with-examples-6728834016d5>

# ITERATOREN

# DAS ITERATOR PROTOKOL

- Iteration = Wiederholung, Iterator = Objekt mit "Datensätzen".
- Ein Objekt ist ein Iterator wenn sein Interface oder seine API zwei Fragen beantworten kann:
- Gibt es noch ein weiteres Element?
- Falls ja, welches?

# DAS ITERATOR PROTOKOL

- Ein Iterator besitzt eine `next()` Methode, die zwei Antworten gibt.
- `done`: zeigt an (`true`, `false`), ob noch weitere Elemente abgerufen werden können.
- `value`: das aktuell gefundene Element.
- Bei jedem Aufruf von `next()` wird der nächste `value` aufgerufen:  
`{ value: 'next value', done: false }`
- Bei `next()` nach dem letzten Element kommt  
`{done: true: value: undefined}`

# DAS ITERABLE PROTOKOL

- Ein Objekt ist iterierbar, wenn es eine Methode `[Symbol.iterator]()` besitzt.
- `[Symbol.iterator]()` hat keine Argumente und gibt einen Iterator zurück.
- `[Symbol.iterator]()` hat den Datentyp `symbol`.

# [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]
```

# [SYMBOL.ITERATOR]()

```
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)  
}
```



<https://codeburst.io/a-simple-guide-to-es6-iterators-in-javascript-with-examples-189d052c3d8e>

# NEUE METHODEN FÜR STANDARDOBJEKTE

# NEUE ARRAY METHODEN

# ARRAY METHODS

## Array

- .from()**
- .isArray()**
- .of()**

## Array.prototype

- .concat()**
- .copyWithin()**
- .entries()**
- .every()**
- .fill()**
- .filter()**
- .find()**
- .findIndex()**
- .flat()**
- .flatMap()**
- .forEach()**
- .includes()**
- .indexOf()**

- .join()**
- .keys()**
- .lastIndexOf()**
- .map()**
- .pop()**
- .push()**
- .reduce()**
- .reduceRight()**
- .reverse()**
- .shift()**
- .slice()**
- .some()**
- .sort()**
- .splice()**
- .toLocaleString()**
- .toSource()**
- .toString()**
- .unshift()**
- .values()**

# OBJECT TO ARRAY

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

for (const x of arrayLikeObject) {           // TypeError
  console.log(x);
}

const arr = Array.from(arrayLikeObject);
for (const x of arr) {                       // OK, iterable
  console.log(x);
}

// Output:
// a
// b
```

# MAP()

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

const names1 = Array.prototype.map.call(
  spanElements,
  s => s.textContent
);

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

# ARRAY CREATING

**Array.of**(item\_0, item\_1, ...)

creates an Array whose elements are item\_0, item\_1, etc.

# ARRAY.FIND(), ARRAY.FINDINDEX()

```
let arr = [ 1, 3, 4, 2 ]  
  
arr.find(x => x > 3)      // 4  
arr.findIndex(x => x > 3) // 2
```

ECMAScript 5

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



# TYPED ARRAYS

```
// create a TypedArray with a size in bytes
const typedArray1 = new Int8Array(8);
typedArray1[0] = 32;

const typedArray2 = new Int8Array(typedArray1);
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

# NEW OBJECT METHOD

## Object

- **assign()**
- 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

```
let dest = { value: 0 },  
    src1 = { foo: 1, bar: 2 },  
    src2 = { foo: 3, baz: 4 };
```

`Object.assign(dest, src1, src2)`

```
// value === 0;  
// dest.foo === 3;  
// dest.bar === 2;  
// dest.baz === 4
```

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

```
let x = 0, y = 0;  
let obj = { x, y };
```

ECMAScript 5

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

# COMPUTED PROPERTY NAMES

```
let obj = {  
  foo: "bar",  
  ["baz" + quux()]: 42  
}
```

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

# METHOD PROPERTIES

```
obj = {  
    foo (a, b) { ... },  
    bar (x, y) { ... },  
    *quux (x, y) { ... }  
}
```

ECMAScript 5

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



# NEUE METHODEN VON STRING, NUMBER, MATH

# STRING METHODS

## String

- .fromCharCode()
- .fromCodePoint()
- .raw()

## String.prototype

- .anchor()
- .big()
- .blink()
- .bold()
- .charAt()
- .charCodeAt()
- .codePointAt()
- .concat()
- .endsWith()**
- .fixed()
- .fontcolor()
- .fontsize()
- .includes()**
- .indexOf()
- .italics()
- .lastIndexOf()
- .link()
- .localeCompare()
- .match()
- .matchAll()
- .normalize()

- .padEnd()
- .padStart()
- .repeat()**
- .replace()
- .search()
- .slice()
- .small()
- .split()
- .startsWith()**
- .strike()
- .sub()
- .substr()
- .substring()
- .sup()
- .toLocaleLowerCase()
- .toLocaleUpperCase()
- .toLowerCase()
- .toSource()
- .toString()
- .toUpperCase()
- .trim()
- .trimRight()
- .trimLeft()
- .valueOf()

# NEUE STRING METHODEN

```
      0 1 2 3 4
let str='hello';

str.startsWith("ello", 1) // true   startet ab Position 1 mit ello
str.endsWith("hell", 5)  // true   endet mit 'hell' vor Position 5
str.includes("ell")      // true   enthält 'ello'
str.includes("ell", 1)   // true   enthält 'ell' ab Position 1
str.includes("ell", 2)   // false  enthält 'ell' ab Position 2
```

```
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.isNaN(42) === false  
Number.isNaN(NaN) === true
```

```
Number.isFinite(Infinity) === false  
Number.isFinite(-Infinity) === false  
Number.isFinite(NaN) === false  
Number.isFinite(123) === 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.isSafeInteger(42) === true  
Number.isSafeInteger(9007199254740992) === 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 FÜR GENAUEREN FLIEßKOMMA VERGLEICH

```
0.1 + 0.2 === 0.3 // false
```

```
Math.abs((0.1 + 0.2) - 0.3) < Number.EPSILON // true
```

# TRUNC

Ganzzahlermittlung:

```
Math.trunc(42.7)    // 42  
Math.trunc( 0.1)    //  0  
Math.trunc(-0.1)    // -0
```

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

# NUMBER SIGN DETERMINATION

Vorzeichenbestimmung:

```
Math.sign(7)      // 1
Math.sign(0)      // 0
Math.sign(-0)     // -0
Math.sign(-7)     // -1
Math.sign(NaN)    // NaN
```

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