- Sammlungen
  - Arrays
  - Map
  - Spread-Operator
  - Destructuring Assignment
- Datum und Zeit
- Asynchrone Funktionen
- SPA: Model

Quellen:

- https://javascript.info/data-types

Arrays

- Zugriff und Ändern von Elementen funktioniert wie in Java, Erzeugen ist anders!

```
// creating arrays
let arr = [];
let fruits = ["Apple", "Orange", "Plum"];

//get an element
console.log( fruits[0] ); // Apple
console.log( fruits[1] ); // Orange
console.log( fruits[2] ); // Plum

//change the value of an element
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

- Anders als in Java kann ein Array beliebig verlängert werden!

```
// change the length of an array
fruits[5] = 'Pineapple'
console.log(fruits.length); // 6
```

Achtung: Zelle mit dem Index 4 hat nun den Wert undefined

- Die Länge eines Arrays lässt sich über die Eigenschaft *length* abfragen.

Arrays

- Ein Array kann beliebige Werte enthalten, sogar Funktionen.

```
// mix of values
let test = [ 'Apple', { name: 'John' }, true, function() { console.log('hello'); } ];

// get the object at index 1 and then show its name
console.log( test[1].name ); // John

// get the function at index 3 and run it
test[3](); // hello


// get the whole array as a string
var text = fruits.toString();
console.log(text); // Apple,Orange,Pear,Lemon
```

Arrays: Methoden

Die Methoden push/shift/unshift erlauben es, ein Array wie eine Queue zu verwenden.

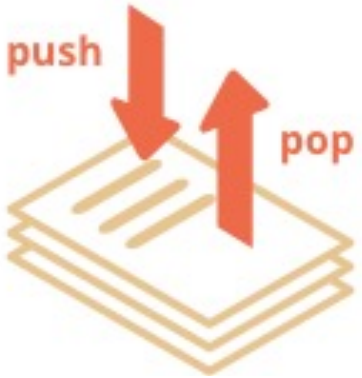
- ***push(element)***: hängt ein (oder mehrere) Element(e) hinten an das Array an
- ***shift()***: holt das erste Element aus dem Array und verschiebt nachfolgende Elemente um eine Zelle nach vorne.
- ***unshift(element)***: fügt ein (oder mehrere) Element(e) vorne in das Array ein und verschiebt alle nachfolgenden Elemente um eine Zelle nach hinten.

```
fruits = ['Apple', 'Orange', 'Pear', 'Lemon', 'Pineapple'];

// shift
console.log( fruits.shift() ); // remove Apple and print it
console.log( fruits ); // Orange, Pear, Lemon, Pineapple

//unshift
fruits.unshift('Grapes', 'Banana');
console.log( fruits ); // Grapes, Banana, Orange, Pear, Lemon, Pineapple
```

Arrays: Methoden



Die Methoden push/pop erlauben es, ein Array wie einen Stack zu verwenden.

- ***push(element)***: hängt ein (oder mehrere) Element(e) hinten an das Array an
- ***pop()***: entfernt das letzte Element aus dem Array.

```
// push
fruits.push('Strawberry', 'Coconut');
console.log( fruits );  // Orange, Pear, Lemon, Pineapple, Strawberry,
                        // Coconut

// pop
var element = fruits.pop();
console.log(element); // Coconut
console.log( fruits );  // Orange, Pear, Lemon, Pineapple, Strawberry
```

Arrays: Methoden

Die Methoden ***splice*** erlaubt es, Zellen aus einem Array zu entfernen bzw. durch Zellen mit neuen Werten zu ersetzen, bzw. neue Zellen mit Werten einzufügen.
Es liefert ein Array mit den entfernten Elementen zurück.

arr.splice(index[, deleteCount, elem1, ..., elemN])

Index ab dem entfernt werden soll

Wie viele Zellen ab index

Beliebige Anzahl an Werten, die ab dem Index eingefügt werden sollen.

Bsp: Entfernen und Einfügen

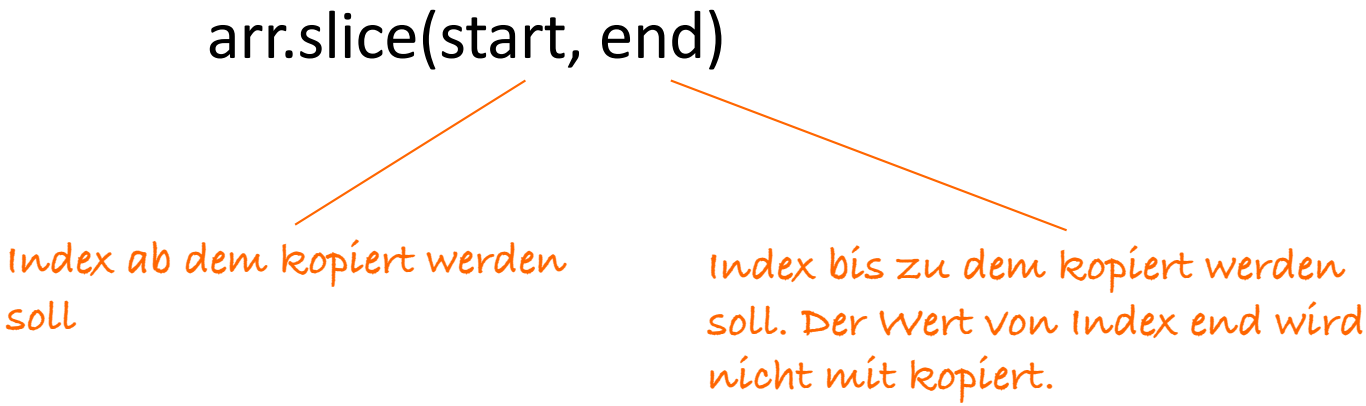
```
let arr = ["I", "study", "JavaScript", "right", "now"];

// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");

console.log( arr ) // now ["Let's", "dance", "right", "now"]
```

Arrays: Methoden

Die Methoden ***slice*** erlaubt es, Teilarrays aus dem Array zu kopieren.



Bsp:

```
let arr = ["I", "study", "JavaScript", "right", "now"];

console.log( arr.slice(1, 3) ); // ["study", "JavaScript"]

// negative Werte werden vom Ende des Arrays aus gezählt
console.log( arr.slice(-2) ); // ["right", "now"]
```

Arrays: Methoden

Die Methoden ***concat*** nimmt Arrays bzw. einzelne Werte entgegen und liefert ein Array zurück, in dem die angegebenen Werte hintereinander gehängt sind.

arr.concat(arg1, arg2, ...)

Beliebige Anzahl an Argumenten,
entweder Arrays oder einzelne Werte

```
arr = [1, 2];

// merge arr with [3,4]
let test1 = arr.concat([3, 4])
console.log(arr);  // [1, 2]
console.log(test1); // [1,2,3,4]

// merge arr with [3,4] and [5,6]
let test2 = arr.concat([3, 4], [5, 6])
console.log(test2); // [1,2,3,4,5,6]

// merge arr with [3,4], then add values 5 and 6
let test3 = arr.concat([3, 4], 5, 6);
console.log(test3); // [1,2,3,4,5,6]
```

Arrays: Methoden

Die folgenden Methoden erlauben es, ein Array zu durchsuchen:

- ***indexOf/lastIndexOf(item, pos)***: sucht nach dem Element *item* beginnend bei *pos* liefert den Index zurück oder -1.
- ***includes(value)***: liefert *true*, wenn es den Wert *value* im Array gibt, sonst *false*.
- ***find/filter(func)***: filtert alle Elemente durch die Funktion *func*, liefert das erste/alle Element(e), für die die Funktion *true* liefert.

```
arr = [1, 0, false, 0];

console.log( arr.indexOf(0)); // 1
console.log( arr.indexOf(false)); // 2
console.log( arr.indexOf(null)); // -1
console.log(arr.includes(5)) // false
console.log( arr.lastIndexOf(0)); // 3
```

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);
console.log(user.name); // John

let someUsers = users.filter(item => item.id < 3);
console.log(someUsers.length); // 2
```

Arrays: Schleifen

- Die gute alte For-Schleife

```
for(let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

- So etwas wie die For-Each-Schleife

```
for(let fruit of fruits) {
    console.log(fruit);
}
```

Achtung: wird die Schleifenvariable mit let definiert, so ist sie nur im Schleifenrumpf gültig!

Map

Eine *Map* funktioniert wie eine *HashMap* in Java. Allerdings lassen sich beliebige Werte/Objekte als Schlüssel oder Wert speichern. Die folgenden Methoden sind verfügbar:

- ***new Map():*** Erzeugt das Map-Objekt.
- ***map.set(key, value):*** speichert ein key/value-Paar.
- ***map.get(key):*** liefert den Wert zu dem Schlüssel oder undefined.
- ***map.has(key):*** liefert true wenn der Schlüssel existiert, sonst false.
- ***map.delete(key):*** entfernt das key/value-Paar für den Schlüssel.
- ***map.clear():*** leert die Map.
- ***map.size:*** liefert die Anzahl der gespeicherten key/value-Paare.

Map: Initialisierung

- Eine Map kann mit Hilfe eines Arrays von key/value-Paaren initialisiert werden

```
let map1 = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);
```

Array mit key/value-Paaren

Achtung: 1 ist ein anderer Key als '1'

```
console.log( map1.get(1)   ); // 'num1'
console.log( map1.get('1') ); // 'str1'
console.log( map1.get(true) ); // 'bool1'
console.log( map1.size ); // 3
```

- Eine Map kann auch aus den Attributen eines Objekts initialisiert werden.

```
let map = new Map(Object.entries({
  name: "John",
  age: 30
}));
```

Object.entries(object) liefert ein Array mit attribut/value-Paaren

Map: Iteration über Elemente

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);
```

- Iteration über alle Schlüssel

Zugriff auf alle Schlüssel

```
for (let vegetable of recipeMap.keys()) {
  console.log(vegetable); // cucumber, tomatoes, onion
}
```

- Iteration über alle Werte

Zugriff auf alle Werte

```
for (let amount of recipeMap.values()) {
  console.log(amount); // 500, 350, 50
}
```

- Iteration über key/value-Paare

```
for (let entry of recipeMap) { // the same as of recipeMap.entries()
  console.log(entry); // cucumber,500 (and so on)
}
```

Spread-Operator

Es gibt Methoden, die beliebig viele Argumente annehmen können. Solche Methoden können mit dem Rest-Operator definiert werden:

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}
```

Rest-Operator, bündelt alle Einzelwerte in einem Array args

Was, wenn man alle Werte für die Funktion in einem Array *values* gespeichert hat?

Aufruf: *sumAll(values)* geht nicht!

In diesem Fall hilft der Spread-Operator, er rekonstruiert aus allen Werten in einem Array einzelne Argumente für einen Funktionsaufruf.

```
let arr = [3, 5, 1];

console.log( sumAll(...arr) ); // 9
```

Spread-Operator, liest aus dem Array die einzelnen Argumente für die Funktion aus.

Destructuring Assignment

Ermöglicht auf einfache Weise die Werte eines Arrays zu „entpacken“ und einer Gruppe von Variablen zuzuweisen.

```
let arr = ["Ilya", "Kantor"]

// destructuring assignment
let [firstName, surname] = arr;

console.log(firstName); // Ilya
console.log(surname);  // Kantor
```

Die Werte aus dem Array werden in der vorgefundenen Reihenfolge den Variablen auf der linken Seite zugewiesen.

```
arr = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
let [, , title] = arr;

console.log( title ); // Consul
```

Durch die Angabe von Kommata können Werte übersprungen werden.

Destructuring Assignment

Funktioniert auch mit den Attributwerten eines Objekts.

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Die Attributwerte des Objekts werden den gleichnamigen Variablen zugewiesen.

Durch die Angabe Attributbezeichner: Variablenname können alternative Variablennamen verwendet werden.

```
let {width: w, height: h, title} = options;
// width -> w
// height -> h
// title -> title
```

Destructuring Assignment

Vorgeben von Defaultwerten

```
options = {
  title: "Menu"
};

let {width = 100, height = 200, title} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Defaultwerte: werden verwendet, wenn kein gleichnamiges Attribut vorhanden ist.

Kobination aus Variablenname und Defaultwert.

```
let {width: w = 100, height: h = 200, title} = options;

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

Datum und Zeit können über den Konstruktor ***Date*** erzeugt werden

- ***Aktuelles Datum***

```
let now = new Date();
console.log(now);
```

- ***Per String***

```
let date = new Date("2017-01-26");
console.log(date);
// Thu Jan 26 2017 01:00:00 GMT+0100 (Mitteleuropäische Normalzeit)

date = new Date("2018-08-28T11:58:00")
console.log(date);
// Tue Aug 28 2018 11:58:00 GMT+0200 (Mitteleuropäische Sommerzeit)
```

T als Begrenzer zwischen Datum und Uhrzeit, Uhrzeit mit „:“.

- ***Mit einzelnen Werten***

```
date = new Date(2018, 9, 24, 11, 20, 0, 0);
console.log(date);
// Wed Oct 24 2018 11:20:00 GMT+0200 (Mitteleuropäische Sommerzeit)

date = new Date(2018, 9, 24);
console.log(date);
// Wed Oct 24 2018 00:00:00 GMT+0200 (Mitteleuropäische Sommerzeit)
```

Achtung! Monatsangabe beginnt bei 0.

Achtung! Standard-Uhrzeit 00:00:00.

Date besitzt die folgenden getter-/setter-Methoden (Auswahl):

- getFullYear() / setFullYear(year [, month, date])
- getMonth() / setMonth(month [, date])
- getDate() / setDate(date)
- getHours() / setHours(hour [, min, sec, ms])
- getMinutes() / setMinutes(min [, sec, ms])
- getSeconds() / setSeconds(sec [, ms])

Zwei Objekte vom Typ ***Date*** können über Vergleichsoperatoren verglichen werden:

```
let date1 = new Date(2018, 9, 24, 11, 20, 0, 0);
let date2 = new Date(2018, 9, 24, 11, 30, 0, 0);

if(date2 > date1) {
    console.log(date2 + " ist größer als " + date1);
}
```

Formatierte Ausgabe über die Methode ***Date.prototype.toLocaleDateString***

dateObj.toLocaleDateString([*locales* [, *options*]])

Länderangabe, z.B. 'de-DE'

Formatierungsangaben für die Darstellung von Jahr, Monat usw.

Optionen:

- ***weekday, month***: narrow, short, long
- ***year, month, day, hour, minute, second***: numeric, 2-digit
- ***timeZoneName***: short, long.

```
date = new Date(2018, 0, 7, 7, 8, 0);

var options1 = {weekday: 'long', year: 'numeric', month: 'long',
                day: 'numeric', hour: 'numeric', minute: 'numeric'};

var options2 = {weekday: 'short', year: '2-digit', month: 'short',
                day: '2-digit', hour: 'numeric', minute: 'numeric'};

let text = date.toLocaleDateString('de-DE', options1);
console.log(text); // Sonntag, 7. Januar 2018, 07:08

text = date.toLocaleDateString('de-DE', options2);
console.log(text); // So., 07. Jan. 18, 07:08
```

Synchrone Funktionen

- Die meisten Funktionen in JavaScript sind synchron, d.h. die Funktionen liefern den berechneten Wert als Rückgabewert an den Aufrufer zurück. Dieser kann sofort damit weiter arbeiten.

```
function producer(value) {
    let result = 10;

    let action = function () {
        result += value;
        return result;
    };

    return action();
}

function consumer(value1, value2) {
    console.log("value1 ist: " + value1);
    console.log("value2 ist: " + value2);
}

function applyProducer() {
    let result1 = producer(11);
    let result2 = producer(12);
    consumer(result1, result2);
}

applyProducer();
```

Berechnung von result.

Rückgabe von result.

Action berechnet Rückgabewert.

Berechnung der Werte durch Producer.

Aufruf des Consumers mit beiden Werten.

Aufruf der Methode.

Asynchrone Funktionen

- Es gibt ein paar Funktionen in JavaScript, die asynchron arbeiten, d.h. die nicht sofort ein Ergebnis zurück liefern bzw. die verzögert ausgeführt werden. Diese Funktionen werden nur angestoßen, danach wird direkt mit dem nächsten Anweisung weiter gemacht. Es wird also nicht auf deren Beendigung gewartet.

```
function producer(value) {
    let result = 10;

    let action = function () {
        setTimeout(() => {
            result += value;
        }, 1000);
        return result;
    };

    return action();
}

function consumer(value1, value2) ...4 lines

function applyProducer() {
    let result1 = producer(11);
    let result2 = producer(12);
    consumer(result1, result2);
}

applyProducer();
```

setTimeout(func, delay) ist eine asynchrone Funktion. Die Funktion func wird mit delay Millisekunden Verzögerung ausgeführt. Auf die Berechnung wird nicht gewartet.

Wird aufgerufen sobald setTimeout angestoßen wurde. D.h. der Wert in result ist immer 10.

Aufruf des producers liefert immer den Wert 10 zurück.

Asynchrone Funktionen: Rückgabewerte über Callback-Funktionen

- Da eine asynchrone Funktion den Wert nicht über eine return-Anweisung zurückgeben kann, kann man ihr eine Callback-Funktion übergeben, die aufgerufen wird, sobald der berechnete Wert vorliegt.

```
function producer(value, callback) {
    let result = 10;
    let action = function () {
        setTimeout(() => {
                result += value;
                callback(result);
            }, 1000);
    };

    action();
}

function consumer(value1, value2) ...4 lines

function applyProducer() {
    producer(11, (result1) => {
        producer(12, (result2) => {
            consumer(result1, result2);
        });
    });
}

applyProducer();
```

Producer bekommt Funktion callback übergeben um das Ergebnis zu übermitteln.

Callback-Funktion wird das Ergebnis übergeben, sobald es berechnet wurde.

Callback-Funktion 1, wird aufgerufen von Producer mit erstem Wert.

Callback-Funktion 2, wird aufgerufen von Producer mit zweitem Wert.

Problem: Callback-Hölle

- Bei einem Aufruf von asynchronen Funktionen dienen Callback-Funktionen der Synchronisation der Rückgabewerte. Bei starken Abhängigkeiten kann dies zu einer unübersichtlichen Schachtelung von Callback-Funktionen führen.

Callback-Hölle:
Sequentialisierung der asynchronen Funktionen durch Callback-Funktionen führt zu einer unübersichtlichen Verschachtelung.

```
function applyProducer() {
    producer(11, (result1) => {
        producer(12, (result2) => {
            producer(13, (result3) => {
                producer(14, (result4) => {
                    consumer(result1, result2, result3, result4);
                });
            });

        });
    });
}
```

Lösung: Promises

- Ein Promise bezeichnet einen Platzhalter für ein Ergebnis, das noch nicht bekannt ist, meist weil seine Berechnung noch nicht abgeschlossen ist. Über Promises können asynchrone Funktionen Ergebniswerte zurückliefern, die Funktionen liefern sozusagen ein Versprechen auf einen zukünftigen Wert.
- Promises können drei Zustände haben: *pending* (in Berechnung), *fullfilled* (Wert liegt vor) oder *rejected* (Berechnung fehlgeschlagen).
- Viele asynchrone Funktionen liefern bereits Promises zurück, die Konstruktorfunktion kann verwendet werden, um asynchrone Funktionen einzupacken, die noch nicht auf Promises umgestellt sind.

```
function producer(value) {
    let result = 10;

    let action = function () {
        let promise = new Promise((resolve) => {
            setTimeout(() => {
                result += value;
                resolve(result);
            }, 1000);
        });
        return promise;
    }

    return action();
}
```

Dem Konstruktor von Promise wird eine Callback-Funktion übergeben, in der die asynchrone Methode aufgerufen wird.

über die resolve-Funktion kann der Promise das Ergebnis mitgeteilt werden, sobald es vorliegt.

action liefert das Promise-Objekt zurück, bevor dieses fulfilled ist.

Der Producer liefert das Promise-Objekt zurück, bevor dieses fulfilled ist.

Verarbeiten von Promises mit *then*

- Auf einem Promise-Objekt lassen sich Methoden aufrufen, um die Verarbeitung der asynchron produzierten Werte zu synchronisieren:
  - ***then(func):*** wird erst ausgelöst, wenn die Promise im Zustand *fulfilled* ist und der Wert vorliegt. Der Wert wird der Funktion *func* übergeben. Die Methode *then* liefert ihr Ergebnis (kann auch *undefined* sein) ebenfalls als Promise zurück, so dass sich die then-Aufrufe verketten lassen.
  - ***catch(func)***: wird ausgelöst, wenn die Promise im Zustand *rejected* ist. In diesem Fall wird der Funktion *func* der Fehler übergeben. Die Methode then liefert eine Promise zurück, so dass sich weitere then-Funktionen anhängen lassen.

```
function applyProducer() {

    let result = 0;

    producer(11)
            .then(result1 => result = result1)
            .then(() => producer(12))
            .then(result2 => consumer(result, result2))
            .catch((error) => console.log(error));
}
```

Sobald die Promise von producer(11) fulfilled ist, wird das Ergebnis an die Callback-Funktion des ersten then-Aufrufs übergeben.

Zwischenergebnis in result speichern.

Vorige Promise liefert keinen Wert, producer aufrufen.

Sobald Promise von producer(12) fulfilled, consumer mit beiden Werten aufrufen.

catch löst aus, wenn eine der Promises rejected ist

Verarbeiten von Promises mit *async / await*

- **Problem** mit *then*: Auch hier kann es leicht unübersichtlich werden, da die *then*-Methoden mit Callback-Funktionen arbeiten.

- ***Besser:*** *explizites Warten auf Werte von asynchronen Funktionen, verarbeiten wie bei der synchronen Auswertung:*

```
async function applyProducer() {
    let result1 = await producer(11);
    let result2 = await producer(12);
    consumer(result1, result2);
}
```

- ***await*:** Operator wartet auf den Ergebniswert einer fullfilled Promise und gibt ihn zurück. Der await-Operator kann nur in einer asynchronen Funktion genutzt werden

- **async*:*** deklariert eine asynchrone Funktion. Der Rückgabewert (oder *undefined*) der Funktion wird selber wieder in eine Promise gepackt. Die Funktion kann daher in anderen Funktionen selber wieder mit *await* aufgerufen werden.

Nebenläufiges Warten auf Promises mit *Promise.all(..)*

- ***Synchrones*** *Warten auf voneinander unabhängige Werte von asynchronen Funktionen kostet unnötig viel Zeit:*

```
async function applyProducer() {
    let result1 = await producer(11);
    let result2 = await producer(12);
    consumer(result1, result2);
}
```

Warten auf result1.
Warten auf result2.
Verarbeiten von result 1 und result2.

- ***Besser***: Anfragen der Werte nebenläufig zueinander starten und warten bis alle vorliegen.

- **Promise.all([p1, p2, p3, …])*:*** bekommt ein Array mit Promises übergeben und liefert ein Array mit den erzeugten Werten zurück. Die Funktionen werden nebenläufig abgearbeitet, es kann gewartet werden, bis alle Ergebnisse vorliegen.

Warten, bis alle Ergebnisse vorliegen.

Destructuring Assignment.

```
// Nebenläufiges Abarbeiten der Aufrufe
async function applyProducer() {
    const [result1, result2] = await Promise.all([producer(11),producer(12)]);
    consumer(result1, result2);
}
```

Model

- Ist für die Datenverwaltung zuständig. Beschafft und konvertiert Daten vom Server. Der Abruf der Daten vom Server erfolgt asynchron.
- Besitzt für jede Information, die der Presenter benötigt, eine asynchrone Funktion, mit der auf die Informationen zugegriffen werden kann sowie Funktionen um Daten anzulegen bzw. zu ändern. Die asynchronen Methoden liefern eine Promise für die angefragten Informationen zurück.
- Darüber hinaus kann das Modul private Funktionen und Variablen enthalten.

```
...

// Öffentliche Schnittstelle des Models

   async function getAll() {
         ...
         return result;
   },

   async getOne(id) {
         ...
         return result;
   },
   ...

export { getAll, getOne, ...};
```

Private Variablen und Funktionen.

Asynchrone Funktion.

Daten vom Server abrufen.

Das Ergebnis wird als Promise zurück geliefert.

Exportieren der öffentlichen Funktionen des Moduls.

Model

- Da die Server-Daten oft zu viele und ungünstig strukturierte Informationen enthalten, sollte das Model daraus geeignete Objekte erzeugen.

```
// Private Elemente des Moduls

function Data(a,b,c) {...}

     ...

// Oeffentliche Schnittstelle des Models

async function getAll() {
   ...
   let arr = [];
   for(let d of result)
      arr.push(new Data(d.a, d.z, d.b));
   return arr;
}

async function getOne(id) {
   ...
   let data = new Data(result.a, result.z, result.b);
   return data;
}
     ...


export { getAll, getOne, ...};
```

Konstruktor für das Erzeugen eines geeigneten Datenobjekts.

Abfrage der Daten vom Server.

Leeres Array für die Objekte anlegen.

Iterieren über alle Rückgabeobjekte, Data-Objekt erzeugen und in das Array schieben.

Neues Data-Objekt aus den Informationen des Rückabeobjekts erzeugen.

Das neue Data-Objekt wird als Promise zurück geliefert.

Bsp: Private Elemente in *model.js*

```
// Private Elemente des Moduls

let mitarbeiter = new Map();
let abteilungen = new Map();
let professions = new Set();
let last_id = 14;
const DELAY = 2000;

function initData() ...25 lines


function Mitarbeiter(id, name, date, job, email, abt) {
    this.id = id;
    this.name = name;
    this.date_of_birth = date;
    this.job = job;
    this.email = email;
    this.abtId = abt;
}

function Abteilung(name, id) ...5 lines

Mitarbeiter.prototype = {...14 lines }

Abteilung.prototype = {...13 lines };

function makePromise(value) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(value);
        }, DELAY);
    });
}
```

Fake-Server: Objekte werden hier lokal verwaltet.

Erstellen von Beispieldaten.

Konstruktor für das Erzeugen von Mitarbeiter-Objekten.

Konstruktor für das Erzeugen von Abteilung-Objekten.

Erweitern der Prototypen um zusätzliche Methoden.

Daten in Pseudo-Promise einpacken, Verzögerung der Bereitstellung der Daten per setTimeout.

Bsp: Öffentliche Schnittstelle in *model.js*

```
// Öffentliche Schnittstelle von Model

async function getAllMitarbeiter(abtId) {
    let alle = [...mitarbeiter.values()];
    let allevonAbt = alle.filter((m) => (m.abtId === abtId));
    return makePromise(allevonAbt);
}

async function getAllAbteilungen() ...3 lines

async function getMitarbeiter(id) {
    let data = mitarbeiter.get(id);
    return makePromise(data);
}

async function getAbteilung(id) ...4 lines

async function deleteMitarbeiter(id) ...9 lines

initData();

export { getAllMitarbeiter,
         getAllAbteilungen,
         getMitarbeiter,
         getAbteilung,
         deleteMitarbeiter
     };
```

Asynchrone Funktion, liefert Promise zurück.

Werte aus der HashMap in Array.

Filtern von allen Mitarbeitern der Abteilung.

Der Rückgabewert wird als Promise geliefert.

Erzeugen der Beispieldaten

Export der öffentlichen Funktionen des Moduls