JavaScript - Fortgeschritten

Web-Engineering II

Prof. Dr. Sebastian von Klinski

JavaScript

- Ist zu einer der wichtigsten Sprachen für Web-Development geworden
- Durch Node.js auch im Backend
- Java und JavaScript haben nur wenig gemein
- Der Umstieg von Java auf JavaScript ist häufig nicht leicht, weil die Syntax zum Teil ähnlich ist, aber…
  - In JavaScript die Typsicherheit fehlt
  - In Java Konzepte wie Funktions-Pointer, Promises, Asynchrone Funktionen und vieles mehr in dieser Form nicht existieren/ nicht gängig sind
  - Arrow-Funktionen eine vollkommen andere Syntax haben
  - Etc.
- In dieser Vorlesung sollen einige wesentliche Unterschiede, bzw. Besonderheiten von JavaScript erläutert werden

ECMAScript

- ECMAScript ist ein Standard
- ECMAScript ist Grundlage für diverse Sprachen: ActionScript, JavaScript, JScript
- ECMAScript: Sub-Set von JavaScript
- Browser unterstützen in der Regel große Teile von ECMAScript, manche jedoch nicht alles! (insbesondere IE und Edge)
- JavaScript…
  - Früher Produkt von Sun, heute von Oracle
  - Ist eine Implementierung von ECMAScript

JavaScript

- Zahlreiche Versionen von ECMAScript
  - ECMAScript 3: Dezember 1999
  - ECMAScript 4: Juli 2008
  - ECMAScript 5: Dezember 2009
  - ECMAScript 6: 2015
  - ECMAScript 7: 2016
  - ECMAScript 8: 2017
  - ECMAScript 9: 2018
- Die meisten grundlegenden Neuerungen mit ECMAScript 6
- Hintergrund:
  - Viele Web-Anwendungen sind heute clientseitige Web-Anwendungen
  - Der Web-Client greift auf ein REST-Backend zu
  - Veröffentlichung von clientseitigen Web-Bibliotheken wie React und Angular

Neue Konzepte mit ECMAScript 6

- Einige Neuerungen sind recht einfach
  - let, const, template strings
  - Rest- und Spread-Operator
- Andere sind etwas komplexer, werden aber häufig genutzt
  - Arrow-Funktionen
  - Klassen
  - Objektliterale
  - Module
  - Promise
- Weitere
  - Generators, map/set/weakmap/weakset, Proxies, Symbols, Tail, Reflection API, etc.

Grundlegende Aspekte

Funktions-Pointer

- In JavaScript kann mit Funktionen umgegangen werden wie mit Objekten
- Sie können
  - Als Variablen definiert werden
  - Als Parameter übergeben werden
  - In Arrays abgelegt werden.
  - Etc.

```
var func = function(a) {
    return a + 1;
}

console.log("Value: " + func(2))
```

- Hinweis: in Funktionen müssen keine Datentypen angegeben werden!
- Seit Java 8 auch in Java über funktionale Interfaces möglich

Quelle: https://www.w3schools.com/js/js_es6.asp

Objekte

- In JavaScript sind Objekte keine Instanzen von Klassen, sondern Sammlungen von Attributnamen und Werten, die im Initialisierer in geschwungenen Klammern geschrieben werden

```
const object1 = { a: 'foo', b: 42, c: {} };
```

Quelle: https://www.w3schools.com/js/js_es6.asp

if

- If-Statements sind in JavaScript weniger strikt definiert als in Java
- Häufig wird if verwendet, um Objekte zu prüfen

```
var object = …
…
if(object) {
…
}
```

- Das if-Statement prüft, ob das Objekt „wahr“ ist…
- Es ist falsch, bei den folgenden Werten
  - Null
  - Undefined
  - NaN
  - Leerer Text (z.B. „“)
  - 0
  - false

Let

- Mit „let“ können Variablen deklariert werden, die nur in diesem Block sichtbar sind

```
 var x = 10;
// Here x is 10
{
  let x = 2;
  // Here x is 2
}
// Here x is 10
```

Quelle: https://www.w3schools.com/js/js_es6.asp

Const

- Mit „const“ können Konstanten deklariert werden, die nur in diesem Block sichtbar sind

```
var x = 10;
// Here x is 10
{
  const x = 2;
  // Here x is 2
}
// Here x is 10
```

Quelle: https://www.w3schools.com/js/js_es6.asp

Template Strings

- Template-Strings sind String-Symbole, die über mehrere Zeilen gehende Zeichenketten sowie eingebettete Javascript-Ausdrücke ermöglichen
- Werden über Accent-Grave eingeschlossen

```
var a = 5;
var b = 10;

console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");

console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

Quelle: https://www.w3schools.com/js/js_es6.asp

Rest-Operator in Funktionen

- Beliebige Anzahl von Parametern können in Array geschrieben und bearbeitet werden

```
function sum(...theArgs) {

    console.log("Args: " + theArgs)

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

  console.log(sum(1, 2, 3));
```

```
PS C:\Beuth-Temp\JavaScriptTest> node .\RestOperator.js
Args: 1,2,3
6
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/rest_parameter

Spread-Operator

- Expandieren eines Ausdrucks (z.B. Arrays) an Stellen, wo mehrere Parameter erwartet werden

```
function myFunction(v, w, x, y, z) { }
var args = [0, 1];
myFunction(-1, ...args, 2, ...[3]);
```

```
var arr = [1,2,3];
var arr2 = [...arr];
```

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Operators/Spread_operator

Klassen

Klassen

- Die Definition von Klassen ist ähnlich wie in Java

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

    print()
    {
        console.log("Rectangle: " + this.width + " / " + this.height)
    }
}

var myRectangle = new Rectangle(3,2)
myRectangle.print()
```

Quelle: https://www.w3schools.com/js/js_es6.asp

Klassen

- Klassen müssen definiert werden, bevor davon Instanzen erzeugt werden
- Das ist bei Funktionen nicht notwendig!

```
var p = new Polygon(); // Fehler!

class Polygon {}
```

- Vererbung funktioniert ähnlich wie bei Java über „extends“

```
class Hund extends Tier{
  sprich() {
    console.log(this.name + ' bellt.');
  }
}
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen

Klassen

- Es gibt mehrere Wege, Klassen zu definieren

```
// Anonyme Klassendeklarartion
var polygon = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};

// Klassendeklaration mit Namen
var polygon = class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen

Klassen

- Klassen werden über Funktionsprototypen umgesetzt und können auch entsprechend dynamisch angelegt werde
- Klassen können auch von Funktionen erben!

```
function Tier() { }

Tier.prototype.sprich = function () {
    console.log("Sprich")
    return this;
}

let obj = new Tier();

obj.sprich()

let sprich = obj.sprich;
sprich(); // Globales Objekt
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen

Objektliterale

Objektliterale

- Ein bisschen wie „Klassen-light“, jedoch ohne Klassendeklaration
- Werden sehr häufig verwendet
- Es können Werte und Funktionen angelegt werden

```
var obj =
{
  eigenschaft : 'blubb',
  methode     : function()
  {
    console.log(this.eigenschaft)
  }
}

obj.methode()
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Klassen

Objektliterale

- Werden sehr häufig für Rückgaben bei Funktionen und Services verwendet
- Direkte Überführung nach Json möglich

```
function getLaptop(make, model, year) {
    return {
        make: make,
        model: model,
        year: year
    }
}

var laptop = getLaptop("Apple", "MacBook", "2015");
console.log("Maker: " + laptop.make)
```

Quelle: https://dev.to/sarah_chima/enhanced-object-literals-in-es6-a9d

Objektliterale

- Über den Spread-Operator (…) können Objektliterale zusammengeführt werden
- Es wird ein neues Objekt erzeugt
- Gleiche Attribute werden überschrieben
- Hinweis: mit JSON.stringify() können Objekte auf Console ausgegeben werden

```
var obj1 = {
    make: "Apple",
    model: "Laptop",
    year: 2020
}
var obj2 = {
    name: "Udo",
    make: "PC"
}
var obj3 = {...obj1, ...obj2}
console.log("Object 3: " + JSON.stringify(obj3))
```

Quelle: https://dev.to/sarah_chima/enhanced-object-literals-in-es6-a9d

Objekt.assign()

- Object.assign() kopiert die Werte aller Objektliterale in das erste Objektliteral
- Es wird das neue Zielobjekt zurückgegeben.
- Gleiche Attribute werden überschrieben
- Hinweis: es wird kein neues Objekt angelegt!

```
var target = { a: 1, b: 2 };
var source = { b: 4, c: 5 };
var returnedTarget = Object.assign(target, source);
console.log(target)
console.log(returnedTarget)
```

```
PS C:\Beuth-Temp\JavaScriptTest> node .\Objektliteral.js
{ a: 1, b: 4, c: 5 }
{ a: 1, b: 4, c: 5 }
PS C:\Beuth-Temp\JavaScriptTest>
```

Arrow-Funktionen

Arrow-Funktionen

- Im Deutschen: Pfeilfunktionen
- Lambda-Ausdrücke, die funktionale Programmierung unterstützen
- Definition von Funktionen mit =>
- Ähnlich wie in C#, Java8 und CoffeeScript
- Verkürzt die Schreibweise von Funktionen deutlich
- Aber: auch Lesbarkeit geht etwas verloren
- Auch in Java möglich

```
// ES5
var x = function(x, y) {
   return x * y;
}

// ES6
const x = (x, y) => x * y;
```

Arrow-Funktionen

- Die Syntax wird bestimmt durch die
  - Anzahl der Parameter, die übergeben werden
  - Die Anzahl der Befehle in der Funktion

```
// Mehrere Parameter und mehrere Befehle
(param1, param2, …, paramN) => { statements }

// Mehrere Parameter und 1 Befehl
(param1, param2, …, paramN) => expression

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

Arrow-Funktionen

- Wenn keine Parameter übergeben werden, müssen runde Klammern gesetzt werden

```
() => { statements }

// Klammern sind optional, wenn nur ein Parametername vorhanden ist:

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

Arrow-Funktionen

- Sehr viele unterschiedliche Varianten für Nutzung
- Map-Funktion von Array: wendet auf jedes Element des Arrays die bereitgestellte Funktion an und gibt das Ergebnis in einem neuen Array zurück.

```
const materials = [  'Hydrogen',  'Helium',  'Lithium',  'Beryllium'];

console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

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

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

Arrow-Funktionen

- Rest Parameter und Default Parameter werden unterstützt

```
(param1, param2, ...rest) => { statements }

(param1 = defaultValue1, param2, …, paramN = defaultValueN) => {
statements }
```

Arrow-Funktionen

- 3 mögliche Umsetzungen mit identischem Ergebnis

```
var elements = [
  "Hydrogen",
  "Helium",
  "Lithium",
  "Beryllium"
];

elements.map(function(element) {
  return element.length
}); // [8, 6, 7, 9]

elements.map(element => {
  return element.length
}); // [8, 6, 7, 9]

elements.map(({length}) => length); // [8, 6, 7, 9]
```

Arrow-Funktionen: this

- In JavaScript definiert jede Funktion sein eigenen this-Kontext
- Im nachfolgenden Code gibt es ein this im Person und ein im setInterval()
- Daher ist this.age in setInterval() nicht definiert

```
function Person() {
    this.age = 0;

    setInterval(function growUp() {
      this.age++;
      console.log("Age: " + this.age)
    }, 1000);
  }

  var p = new Person();
```

```
PS C:\Git\de.beuth.svk.l
Age: NaN
Age: NaN
Age: NaN
Age: NaN
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen

Arrow-Funktionen: this

- In ECMAScript 5 konnte das über eine lokale Variable behoben werden
- Die Umsetzung ist aber nicht wirklich gut

```
function Person() {
    that = this;
    this.age = 0;

    setInterval(function growUp() {
      that.age++;
      console.log("Age: " + that.age)
    }, 1000);
  }

  var p = new Person();
```

```
PS C:\Git\de.beuth.svk.
Age: 1
Age: 2
Age: 3
Age: 4
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen

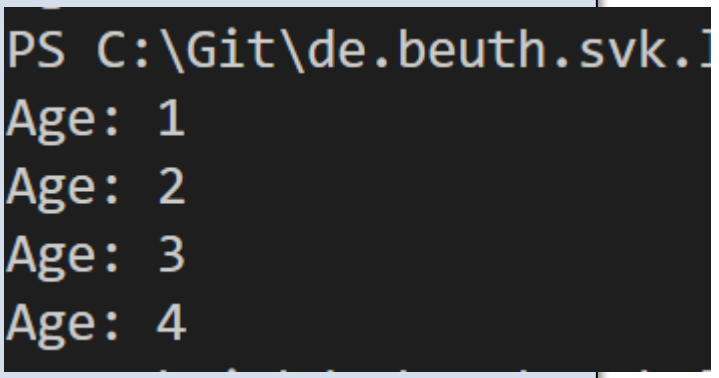
Arrow-Funktionen: this

- Array-Funktionen definieren kein eigenen this-Kontext
- Daher kann die Arrow-Funktion auf das this von Person zugreifen
- Arrow-Funktionen haben keinen eigenen this-Kontext
- Sie verwenden den this-Kontext des Umgebungskontextes

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++;
    console.log("Age: " + this.age)
  }, 1000);
}

var p = new Person();
```



Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen

Arrow-Funktionen: this in Java

- Lambda-Ausdrücke behandeln this-Kontext anders als Interface-Instanzen!

```
private String value = "In Lambda-Klasse";
...
Foo fooIC = new Foo() {
    String value = "Inner class value";
    @Override
    public String method(String string)
    {
       return this.value;
    }
};
String resultIC = fooIC.method("");

Foo fooLambda = parameter -> {
    String value = "Lambda value";
    return this.value;
};
String resultLambda = fooLambda.method("");
System.out.println("Results: resultIC = " + resultIC + ", resultLambda = " + resultLambda);
```

```
@FunctionalInterface
public interface Foo
{
   String method(String string);
}
```

```
<terminated> LambdaTest [Java Application] C:\EigeneProgramme\jdk-13.0.2\bin\javaw.exe (08.10
Results: resultIC = Inner class value, resultLambda = In Lambda-Klasse
```

Arrow-Funktionen

- 3 mögliche Umsetzungen mit identischem Ergebnis

```
var elements = [
  "Hydrogen",
  "Helium",
  "Lithium",
  "Beryllium"
];

elements.map(function(element) {
  return element.length
}); // [8, 6, 7, 9]

elements.map(element => {
  return element.length
}); // [8, 6, 7, 9]

elements.map(({length}) => length); // [8, 6, 7, 9]
```

Arrow-Funktionen

- Arrow-Funktionen sollten eher nicht in Objektliteralen angewandt werden, weil der this-Kontext unklar ist

```
var obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log( this.i, this)
  }
}
obj.b(); // gibt undefined, Window {...} aus (oder das globale Objekt)
obj.c(); // gibt 10, Object {...} aus
```

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Functions/Pfeilfunktionen

Module

Module in JavaScript

- Es gibt unterschiedliche Arten von Modulen, die von Laufzeitumgebung abhängen
- Native JavaScript Module
  - Definiert in ECMAScript 6
  - Teil der Programmiersprache
  - Einbinden über „import“
  - Export über „export“
  - Modul wird nur einmal instanziiert (→ Singleton)
- CommonJS Module
  - Für node.js
  - Serverseitig
  - Einbinden über „require“
  - Export über „module.exports“
  - Es werden Kopien von den Objekten zurückgegeben
- Weitere: Async Module Definition, Universal Module Definition

Native Module

- Unterteilen von JavaScript-Funktionen in Module möglich
- Unterteilung von Modulen auf unterschiedliche Dateien
- Module können Konstanten oder Funktionen exportieren
- Mit „import“ können exportierte Referenzen importiert werden

```
export const name = "Manfred";

export function meinAlter()
{
    return 20;
}
```

```
import {name, meinAlter} from "./Testmodule.js"

console.log("Name: " + name);
console.log("Alter: " + meinAlter());
```

Native Module

- Wenn ein Modul mehrfach importiert wird, wird der Code nur einmal ausgewertet und ausgeführt
- Danach erhalten alle die gleichen Export-Referenzen → wie ein Singleton
- Achtung: potentielle Seiteneffekte!

```
export let admin = {
  name: "John"
};
```

```
import {admin} from './admin.js';
admin.name = "Pete";
```

```
import {admin} from './admin.js';
alert(admin.name); // Pete

// Beide verwenden die gleiche Referenz, Änderungen werden in beiden
// Modulen übernommen
```

Native Module

- Singleton-Eigenschaft von Modulen hat Vor- und Nachteile
  - Zentrale Konfigurationsmodule nutzen gleiche Daten
  - Aber Objekte können von allen geändert werden
  - Schutz der Daten muss über Funktionen umgesetzt werden (keine Objektreferenzen exportieren)
- Statische Prüfung der Exports (zur Compile-Zeit)
- Damit in Node.js natives JavaScript-Module geladen werden können, muss Projekt ein Modul sein
- Dazu muss in package.json folgender Eintrag vorhanden sein.
- Anmerkung: ES6-Module sind derzeit unter node.js „experimental“

```
{
  "type": "module"
}
```

```
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest> node .\Runner.js
(node:16368) ExperimentalWarning: The ESM module loader is experimental.
Name: Manfred
Alter: 20
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest>
```

CommonJS Module

- CommonJS war Projekt, um Konventionen zur Definition von Modulen in JavaScript festzulegen
- War noch vor ECMA6
- Browser unterstützen diese Module nicht, müssen vorher über einen Transpiler in Standard-JavaScript überführt werden
- Werden vor allem in Node.js verwendet
- Erkennbar an require() und module.exports (im Vergleich zu ES6-Modulen)

CommonJS Module

- Module werden als Objekte exportiert (in diesem Fall Funktionsprototyp)

```
const name = "Manfred";

function meinAlter()
{
    return 20;
}

module.exports = {name, meinAlter}
```

```
const modul = require('./NodeModule')

console.log("Name: " + modul.name);
console.log("Alter: " + modul.meinAlter());
```

CommonJS Module

- Es gibt zwei Varianten zum Exportieren von Objekten: exports und module.exports

```
const name = "Manfred";
exports = name;

function meinAlter()
{
    return 20;
}
module.exports = {name, meinAlter}
```

- module ist ein einfaches JavaScript-Objekt, dass für den Zugriff auf das Modul angelegt wird
- exports ist eine JavaScript-Variable, der jede Referenzen zugewiesen werden können, die das Modul zurückgeben soll
- Wenn beides verwendet wird, entscheidet module.exports, was tatsächlich zurückgegeben wird → verwenden Sie eher module.exports

CommonJS Module

- module.exports kann Attribute und Funktionen als Objektliteral zurückgeben
- Dabei können auch Namen für die exportierten Objekte vergeben werden

```
const name = "Manfred";

function meinAlter()
{
    return 20;
}

module.exports = {
    meinName: name,
    meinAlter: meinAlter
}
```

```
exports.a = 1
exports.b = 2
exports.c = 3
```

- Beim Import können auch die Referenzen explizit importiert werden

```
const { a, b, c } = require('./uppercase.js')
```

CommonJS Module

- Wenn diese Module verwendet werden, darf im package.json nicht das Modul-Attribut enthalten sein!
- Ansonsten kompiliert die Anwendung nicht

```
{
  "type": "module"
}
```

Asynchrone Aufrufe

Node.js: asynchrone Aufrufe

- Bei Programmschritten, die länger dauern können, sind synchrone Aufrufe problematisch
  - Blockieren die gesamte Anwendung
  - Prozesse können nicht verteilt werden auf mehrere CPUs
- Daher werden aufwändige oder länger laufende Verarbeitungsschritte asynchron ausgeführt
  - Asynchrone Callback-Funktionen
  - Promise
  - Asynchronen Funktionen

Callback-Funktionen

- Funktion wird aus einer Funktion heraus aufgerufen

```
let meinCallback = function () {
   console.log ("in Callback");
}

let aufrufendeFunktion = function (callback) {
   console.log("aufrufende Funktion");
   callback();
}

aufrufendeFunktion(meinCallback);
```

```
aufrufende Funktion
in Callback
```

- Callback-Funktion ist zunächst nicht asynchron
- Callback-Funktionen werden jedoch häufig an asynchrone Funktionen übergeben

Callback-Funktionen

- Beispiel: Starten eines Servers mit Callback-Funktion

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);

console.log(„Hallo“)
```

- createServer startet einen Server und bekommt eine Callback-Methode übergeben, die alle Anfragen an den Server bearbeitet
- Immer wenn eine Seite vom Server angefragt wird, werden das Request- und Response-Objekt an die Methode übergeben
- Die Konsolenausgabe „Hallo“ unmittelbar nach dem Ausführen des Programms ausgegeben, noch bevor der Server gestartet wurde

Callback-Funktionen

- Beispiel: Anhängen von Text an eine Datei

```
var fs = require('fs');

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
  if (err)
    throw err;
  console.log('Saved!');
});
```

- appendFile bekommt den Dateinamen der zu ändernden Datei, den anzuhängenden Text und eine Callback-Funktion, die ausgeführt wird, wenn das Anhängen des Textes abgeschlossen ist
- Wenn beim Anhängen des Textes ein Fehler aufgetreten ist, wird der Fehler an die Funktion übergeben
- Ansonsten ist das Fehlerobjekt leer

Promise

- Häufig werden Callback-Methoden in Verbindung mit Promise verwendet
- Umsetzung von asynchronen, langlaufenden, schrittweisen Prozessen über klassische Ansätze (z.B. Event-Listener) sehr aufwändig
- Beispiel: Aufrufen einer URL auf einem Server

```
Const url = „http://www.restservice.de/anfrage“;
const xhr = new XMLHttpRequest(url);
xhr.addEventListener ("readystatechange", (url) => {
  if (xhr.readyState === 4) {
    console.log (xhr.responseText);
  }
})
xhr.open ("GET", url);
xhr.send();
```

- Es wird ein Request angelegt und ein Event-Listener angehangen, der aufgerufen wird, wenn das Ergebnis da ist
- Anschließend wird die Verbindung geöffnet und die Anfrage abgeschickt

Promise

- Verwendung von Event-Listener für einzelne Abfrage noch ok, für Sequenzen von Anfragen schnell unübersichtlich

```
const url = "../fetch.json";
const url2 = "../file.json";
const url3 = "../data.json";
const xhr = new XMLHttpRequest(url);
xhr.addEventListener ("readystatechange", (url) => {
  if (xhr.readyState === 4) {
    console.log (xhr.responseText);
    const xhr2 = new XMLHttpRequest(url2);
    xhr2.addEventListener ("readystatechange", (url2) => {
    if (xhr2.readyState === 4) {
      const xhr3 = new XMLHttpRequest(url3);
      xhr3.addEventListener ("readystatechange", (url3) => {
      });
    …
    }
  });
  xhr2.open ("GET", url2);
  xhr2.send();
}});
xhr.open ("GET", url);
xhr.send();
```

Promise

- Ineinander verschachtelte Callback-Methoden werden schnell unübersichtlich → man nennt es „Callback-Hell“
- Promise: Objekte, die das Ergebnis einer asynchronen Aktion abwarten und dann zurückgeben, wenn die Daten da sind oder ein Fehler aufgetreten ist
- Vergleichbar zu Future in Java

Promise

- Einfaches Beispiel:
- Promise wird sofort gestartet (erste Ausgabe),
- setTimeout blockt den Callback für 1 Sekunde, Ausführung wird fortgesetzt (zweite Ausgabe außerhalb des Promise)
- Nach einer Sekunde schließt der Promise ab mit resolve

```
let promise = new Promise(function (resolve, reject) {
    console.log("Im Promise")
    setTimeout(() => {
        console.log("Now resolve it.")
        resolve("done")
    },1000);
});

console.log("Nach dem Promise")
```

```
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest> node .\Promise.js
Im Promise
Nach dem Promise
Now resolve it.
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest>
```

Promise

- Um Anwendung nach Abschluss des Promise weiterzuführen, können per then() ein Callback für den Erfolgsfall und einen für den Fehlerfall übergeben werden

```
var testFunktion = function (testZahl) {
    return new Promise(function (resolve, reject) {
        console.log("Im Promise")
        setTimeout(() => {
            if (testZahl == 1) {
                resolve("done")
            }
            else {
                reject("Ist nicht 1")
            }
        }, 1000);
    }).then(
        () => console.log("Success"),
        () => console.log("Fehler")
    );
}

testFunktion(2)
```

Promise-Verkettung

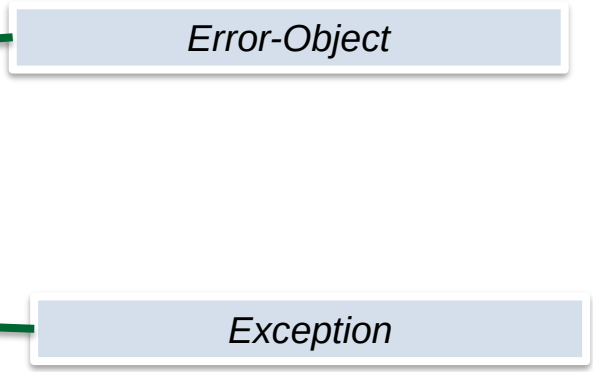
- Promises können über then() miteinander verkettet werden, um immer wieder asynchrone Aufrufe (fetch()) auszuführen

```
fetch (url).then ( function (response) {
    return response.json();
  }).then ( function (data) {
    console.log ("url 1 " + data)
  }).then (fetch (url2).then (function (response) {
    return response.json();
  }).then (function (data) {
    console.log ("url 2 " + data)
  }).then (fetch (url3).then (function (response) {
    return response.json();
  }).then (function (data) {
    console.log ("url 3 " + data)
  })
))
```

Promise: Error-Handling

- Error-Handling in JavaScript ähnlich wie in Java
  - Es werden Fehlerobjekte (Error-Object) mit throw geworfen
  - Es werden Exceptions per catch() gefangen

```
function myApiFunc(callback) {
try {
    doSomeAsynchronousOperation((err) => {
      if (err) {
        throw (err);
      }
      /* continue as normal */
    });
  } catch (ex) {
    callback(ex);
  }
}
```



Quelle: https://www.joyent.com/node-js/production/design/errors

Promise: Exceptions

- Tritt bei einem Promise eine Exception auf, wird diese nicht durch ein synchrones try-catch gefangen

```
function example() {
    return new Promise((resolve, reject) => {
      throw „Fehler“
    });
  }

  try {
    example().then(r => console.log(`.then(${r})`));
  } catch (e) {
    console.error(`try/catch(${e})`);
  }
```

```
(node:12052) UnhandledPromiseRejectionWarning: test reject
(node:12052) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated …
(node:12052) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In …
```

Exception-Handling bei Promise

- Bei einem Promise muss die Catch-Methode vom Promise verwendet werden

```
function example() {
    return new Promise((resolve, reject) => {
      reject("test reject");
    });
  }

  try {
    example()
      .then(r => console.log(`.then(${r})`))
      .catch(e => console.error(`.catch(${e})`));
  } catch (e) {
    console.error(`try/catch(${e})`);
  }
```

Promise-Verkettung

- Exception-Handling sauberer als bei Callbacks

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

- Bei Callback müssten ineinander gestaffelte if-Statements verwendet werden mit jeweils separaten Catch-Blöcken
- Then liefert wieder ein Promise zurück
- Gestaffelte Promise reichen das Reject hoch → nur ein Catch reicht

Exception-Handling bei Promise

- Alternativ zum catch() kann auch beim then() ein Error-Callback verwendet werden

```
var testFunktion = function (testZahl) {
    return new Promise(function (resolve, reject) {
        if (testZahl == 1) {
            resolve("done")
        }
        else if (testZahl == 2) {
            throw new Error("Ein Fehler!")
        }
        else {
            reject("Ist nicht 1")
        }
    }).then(
        () => console.log("Success"),
        () => console.log("Fehler")
    ).catch(err => console.log("Warum denn der Fehler"))
}

testFunktion(2)
```

Promise: Exception-Handling

- Insbesondere bei verschachtelten then()-Aufrufen sollte zum Schluss ein catch() angehangen werden
- Fehler werden von den Promises weitergereicht bis zum ersten catch()
- Innerhalb der Promises sollten eher keine Try-Catch-Blöcke verwendet werden
- Es muss unterschieden werden, welche Exception synchron und welche asynchron geworfen werden können
- Bei synchronen Exceptions muss der Promise-Aufruf noch in einen Try-Catch-Block

```
var testFunktion = function (testZahl) {
    throw new Error("vor dem promise")
    return new Promise(function (resolve, reject) {
        if (testZahl == 1) {
            resolve("done")
        }
        …
```

Async-Funktionen

- Der Umgang mit Promise und Callback-Funktionen ist für viele nicht so angenehm
- Mit Async-Funktionen und „await“ können asynchrone Aufrufe wie synchrone Aufrufe verwendet werden

```
function resolveAfter1Second() {
    return new Promise(resolve => {
        setTimeout(() => {
            console.log("Im Promise")
            resolve('resolved');
        }, 1000);
    });
}

async function asyncCall() {
    console.log('Vor der Asyn Funtion');
    const result = await resolveAfter1Second();
    console.log("Nach dem Async: " + result);
}

asyncCall();
```

```
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest>
Vor der Asyn Funtion
Im Promise
Nach dem Async: resolved
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest>
```

Async-Funktionen

- Async-Funktionen geben immer ein Promise zurück!
- Um Rückgabewerte auszuwerten, muss ein then()-Block verwendet werden

```
async function hello()
{
    return "Hello"
};

console.log("Result: " + hello())

hello().then((value) => console.log("Return value: " + value))
```

```
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest> node
Result: [object Promise]
Return value: Hello
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\JavaScriptTest>
```

Async-Funktionen

Beispiele für die Definition und Nutzung von Async-Funktionen

- Funktionsvariable

```
let hello = async function() { return "Hello" };
hello();
```

- Pfeilfunktion

```
let hello = async () => { return "Hello" };
```

- Rückgabewert über then() ausgeben

```
hello().then((value) => console.log(value))
hello().then(console.log)
```

Async-Funktionen: Exception-Handling

- Da Async-Funktionen ein Promise zurückgeben und asynchron ausgeführt werden, kann kein try-catch verwendet werden, sondern wird das catch() an das Promise angehangen

```
var testFunktion = function (testZahl) {
    return new Promise(function (resolve, reject) {
        console.log("Do the long process")
        reject("Hier ist ein Fehler")
    }).then(
        () => {return "Done"}
    )
}

async function hello() {
    let ergebnis = await testFunktion(1)
    return ergebnis
};

hello().then(
    (value) => console.log("Return value: " + value)
).catch(
    e => console.log(e)
)
```

```
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\Java
Do the long process
Hier ist ein Fehler
PS C:\Beuth-Temp\Vorlesungen\EWD\RestTestProject\Java
```

Async-Funktionen: Exception-Handling

- Wenn mehrere langlaufende Funktionsaufrufe ausgeführt und abgewartet werden sollen, kann Promise.all() verwendet werden

```
async function displayContent() {
  let coffee = fetchAndDecode('coffee.jpg', 'blob');
  let tea = fetchAndDecode('tea.jpg', 'blob');
  let description = fetchAndDecode('description.txt', 'text');

  let values = await Promise.all([coffee, tea, description]);
  …
```

Async-Funktionen

- Async-Funktionen erleichtern auf den ersten Blick den Code, weil asynchrone Funktion aussehen wie synchrone Funktionen
- Doch der Nachteil ist, dass die eigene Funktionsausführung geblockt wird → das kann die Performance der eigenen Anwendung beeinträchtigen
- Wenn das Promise gar nicht antwortet (weil der Service offline ist) kann die Anwendung blockiert werden