Toolbox Javascript

Keywords

MDN Reference

JavaScript Verhalten

JavaScript ist eine interpretierte Sprache. Somit kann man in JavaScript auch keine Methoden überladen.

```
// Methode wird überschrieben.
function fun2()    { return 1; }    //1.
function fun2(arg) { return arg; } // Überschreibt 1.

//Tests
document.writeln( fun2() !== 1  );
document.writeln( fun2() === undefined );
document.writeln( fun2(42) === 42 );
```

return Keyword nicht vergessen

```
function noReturn()    { 1; }
const noReturn2 = () => {1; }; //Daumenregel: Mit geschweiften Klammern braucht
es immer return
const noReturn3 = () => 1;

document.writeln( noReturn() !== 1 );
document.writeln( noReturn2() !== 1 );
document.writeln( noReturn3() === 1 );
```

Curied functions

```
const plus = x => y => x+y

plus(10)(20) === 30 // true
```

Canvas

Canvas Tutorial MDN

```
<html><head><title>Ball</title></head><body onload="start()">

<!-- Create a Canvas Element in the HTML File -->
<canvas id="canvas" width="400" height="400"> </canvas>
</body>
<html>
```

```
//Java Script
const canvas = document.getElementById("canvas");
const context = canvas.getContext("2d");

context.fillStyle = "black";
context.fillRect(0,0,canvas.width, canvas.height);
```

Key Events

```
const rightArrow =39;
const leftArrow = 37;
window.onkeydown = evt =>{
    (evt.keyCode == rightArrow) ? ... : ...;
};
```

Wiederkehrende Aktion setInterval()

```
// Führt Funktion alle 3 Sekunden aus
setInterval( () => alert("Hello"), 3000);

// Ladet neues Game jede 1000/10ms
setInterval(() => {
        console.log(`Snake length : ${snake.length}`)
        nextBoard();
        display(context);
    }, 1000 / 10);
}
```

Arrays in JavaScript

```
// Neues Array erzeugen
let arr1 = [1,2,3];

//Snake Variante
let arr2 = [
    {x: 10, y: 5},
    {x: 10, y: 6},
    {x: 10, y: 7}
];

arr1.pop(); //returns and remove the last Element of the Array [1,2]
arr1.push(3);// adds a new Element at the End of the Array [1,2,3]
```

```
arr1.unshift(0); //put Element at the beginning of the array [0,1,2,3]
arr1.shift(); //returns and removes first elemnt of Array
```

Array deconstructor

```
const foo = () => [1,2];
foo() //[1,2]
// Elemente aus einem Array "holen"
let [x,y] = foo()  // x = 1 | y = 2

// swap Element in Array
[y, x] = [x, y] // x = 2 | y =1
// Swap als Funktion
const swap ([x, y]) => [y, x]
```

Scopes

Every Variable has his own scope

`global` Window (in Browser)

`function` No matter where defined, variables are local to the enclosing function (lambda)

JavaScript Variables

```
x = ...          //mutable, global scope --> Don't use
var x = ...      //mutable, "hoisted" scope --> Don't use
let x = ...      //mutable, local scope
const x = ...    //immutable, local scope
```

IIFE

Immediately Invoked Function Expression

```
( () => {let x = 2; document.writeln(x ===2)}) ()
```

Lambda Kalkül

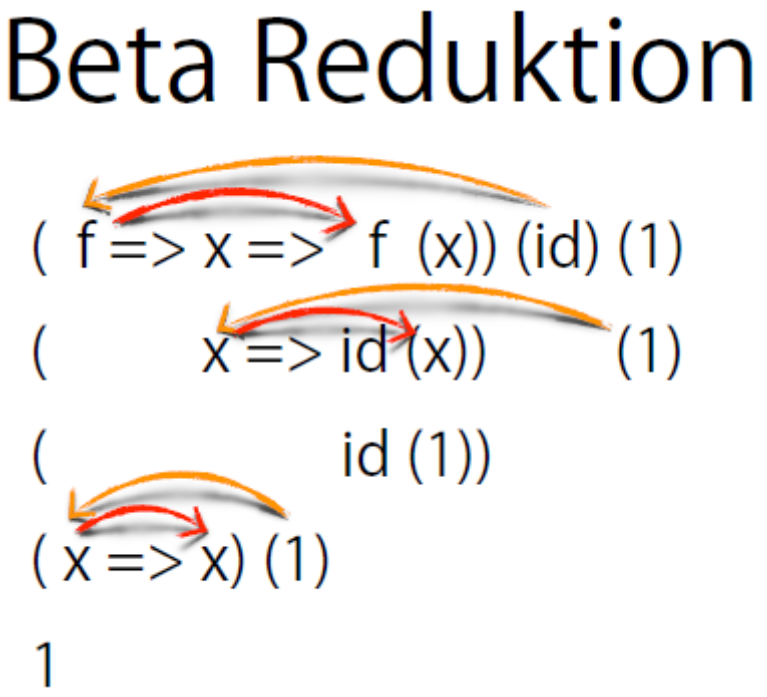
Kalkül = Eine Art, wie man Schlussfolgerungen zieht. --> Logik

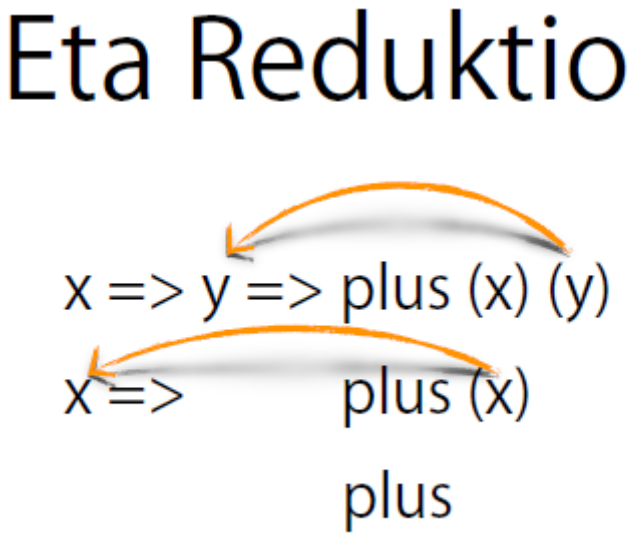
```
//Alpha Translation --> Parameter Umbenennen
const id = x => x
const id = y => y

// Beta Reduktion --> Argument einsetzen
( f => x => f (x))(id)(1)
(      x =>id(x))     (1)
(           id(1))
(x=>x)(1)
1

//Eta Reduktion --> Parameter kürzen
x => y =>plus(x)(y) //Wenn y so da steht
x =>     plus(x)
```

```
        plus
```





Logic with Lambda

Basic Functions

```
const id = x =>x;
const konst = x => y => x;
const fst = konst;
const snd = x => y => y;
```

True | False

```
//Implement True
//const T = a => b => a
const T = fst

//Implement False
//const F = a => b => b
const F = snd;
```

And

```
// And  --> Wenn a True ist gebe B zurück, wenn a False ist gebe a zurück ==
false
const and = a => b => a (b) (a);
```

Or

```
//OR --> Wenn a True ist gebe a zurück (== True) wenn a False ist gebe b zurück
(T|F)
const or = a => b => a(a)(b)
```

Pair

```
//Immutable. f ist wie einen getter und wird mit firstname oder lastname
aufgerufen.
const pair = x => y => f => f (x) (y)
const firstname = fst;
const lastname = snd;
```

regEx

Use Regual Expressions to find or edit patterns in a string.
The literal notation's parameters are enclosed between slashes and do not use quotation marks.

```
let string = "I like Javascript"
// literal notation --> search for the term like and replace it by the word love
string = string.replace(/like/g,'love'); //"I love Javascript"
```

Man kann regex als String angeben oder als regex Object

```
const s = "The rain in Spain stays mainly in the plains";
r1 = s.replace(/ain/g,  "___") ;//replace all the occurrences of "ain"
//The r___ in Sp___ stays m___ly in the pl___s
r2 = s.replace(/\w+/g, "*") // Alle Worte durch Sterne ersetzen
//* * * * * * * * *
document.writeln(r)
```

... Spread Operator

Spread Operator MDN

```
//...y "sammelt" alle Argumente, welche nach x kommen in der Variable y
const f1 = (x, ...y) => console.log(x,y);
f1(1,2) //1-2
f1(1,2,3,4,5,6) //1-[2,3,4,5,6]
//Kapselt Argumente in Y und "entkapselt" sie wieder
const f2 = (x, ...y) => console.log(x, ...y);
f2(1,2,3,4,5,6) //1-2-3-4-5-6
```

High Order Function

Können auf Datenstrukturen angewendet werden

Map

```
const times = a => b => a * b;
const twoTimes = times(2); //partial application --> wartet noch auf "b"

[1, 2, 3].map(x => times(2)(x));
[1, 2, 3].map(times(2));
[1, 2, 3].map(twoTimes);
```

filter

*Gibt an, welche Werte behalten werden. Alle Conditions, welche zu **true** evaluieren, werden behalten*

```
const odd = x => x % 2 === 1;

[1, 2, 3].filter(x => x % 2 === 1);
[1, 2, 3].filter(x => odd(x));
[1, 2, 3].filter(odd);
```

reduce

Die `reduce()`-Methode reduziert ein Array auf einen einzigen Wert, indem es jeweils zwei Elemente (von links nach rechts) durch eine gegebene Funktion reduziert.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

// 1 + 2 + 3 + 4
console.log(array1.reduce(reducer));
// expected output: 10

// 5 + 1 + 2 + 3 + 4
console.log(array1.reduce(reducer, 5));
// expected output: 15

//join an Array with special delimeter
const join = delimeter => (accu, cur) => (accu + delimeter + cur);
[1,2,3].reduce(join('-')); // Output: '1-2-3'
```

Scripting

Progressive Web App

```
document.write('<script src=...');
```

Change Content of Page

```
//change title of Page
document.getElementsByTagName("title")[0].textContent = "New Title"
//STring ausführen mit eval
const i = "alert('hi!');" //save string as Variable
eval(i);    //execute String with eval
```

Function

`Function()` is like `eval()` but declares parameters and executes in the global scope. It creates a reference.

```
const add = Function('x', 'y', 'return x+y');
add(1, 2);
add(2, 3); // no need to re-parse
```

Plotter

Show Plot

```
const userFunction = document.getElementById('user_function');
//with eval
display(context, x => eval(userFunction.value));
userFunction.onchange = evt => display(context, x => eval(userFunction.value));

//with Function
const f = Function('x', `return ${userFunction.value}`);
display(context, f);
userFunction.onchange = evt => display(context, Function('x', `return
${userFunction.value}`));
```

Object Oriented Programming in JS

Objects

What are Objects in JS

- Data Structures
- Method for access and management
- A location for mutual state
- abstraction and polymorphism

Open & dynamic

Standard JS Object

```
const good ={
    firstname : "Good",
    lastname : "Boy",
    getName: function(){
        return this.firstname + " " + this.lastname
    }
};
// no safety but super dynamic
// unobvious how to share structure
// beware of "this"! See Adam Breindl last week.
```

Closed & explicit

Closure scope, no "this"

```
function Person(first, last) {
    let firstname = first; // optional
    let lastname = last;
    return {
        getName: function() {
            return firstname + " " + lastname }
        }
    }
}
// best safety, easy to share structure, but no class
```

Mixed & classified

Depends on "new". Is the "default" construction.

```
const Person = ( () => { // lexical scope
    function Person(first, last) { // ctor, binding
        this.firstname = first;
        this.lastname = last;
    }
    Person.prototype.getName = function() {
        return this.firstname + " " + this.lastname;
    };
    return Person;
}) (); // IIFE
// new Person("Good", "Boy") instanceof Person
```

Example for Player Object

```
function Player() {
    let fallbackIndex = 0;
    let progressIndex = 0;
    return{
        proceed: function(stride){
            progressIndex +=stride;
        },
        fallback: function(){
```

```
            progressIndex = fallbackIndex;
        },
        turn: function(){
            fallbackIndex = progressIndex;
        },
        getFallbackIndex: function(){
            return fallbackIndex;
        },
        getProgressIndex: function(){
            return progressIndex;
        }
    }
}
```

Classes

`Class` Keyword: Syntactic sugar for Mixed - classified (ES6)

```
class Person {
    constructor(first, last) {
        this.firstname = first;
        this.lastname = last
    }
    getName() {
    return this.firstname + " " + this.lastname
    }
}
// new Person("Good", "Boy") instanceof Person
```

`extends` Keyword Syntactic sugar for creating a prototype chain (ES6)

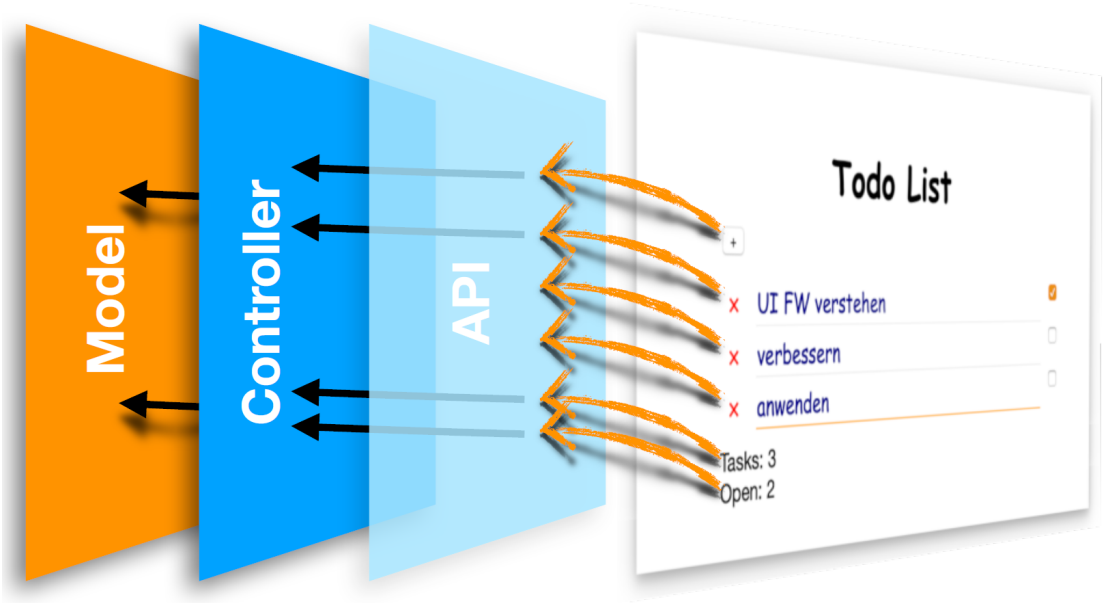
```
class Student extends Person {
    constructor (first, last, grade) {
        super(first, last); //Do not forget !! Superconstructor
        this.grade = grade;
    }
}
const s = new Student("Top","Student", 5.5);
```

Prototype Chain

```
const s = new Student()
// s.__proto__ === Student.prototype;
// Object.getPrototypeOf(s) === Student.prototype;
// => s instanceof Student
```

MVC

MVC, classic version



Higher-Order-Function

```
function test(name, callback) {
    const assert = Assert();        // prework
    callback(assert);               // callback
    report(name, assert.getOk());   // postwork
}
```

Observable

```
const Observable = value => {
    const listeners = []; // many
    return {
        onChange: callback => listeners.push(callback),
        getValue: () => value,
        setValue: val => {
            if (value === val) return; // protection
            // ordering
            value = val;
            listeners.forEach(notify => notify(val));
        }
    }
};
```

Async Programming

Callback, Events

```
function start() {
    //...
    window.onkeydown = evt => {
        // doSomething();
    };
    setInterval(() => {
        // doSomething();
    }, 1000 / 5);
}
```

Promise

Promise

Das `Promise`-Interface repräsentiert einen Proxy für einen Wert, der nicht zwingend bekannt ist, wenn der Promise erstellt wird. Das erlaubt die Assoziation zwischen *Handler* und dem Gelingen oder Fehlschlagen einer asynchronen Aktion. Mit diesem Mechanismus können asynchrone Methoden in gleicher Weise Werte zurück geben wie synchrone Methoden: Anstelle des endgültigen Wertes wird ein *Promise* zurückgegeben, dass es zu einem Zeitpunkt in der Zukunft einen Wert geben wird.

Most prominent use:

```
fetch ('http://fhnw.ch/json/students/list')
.then(response => response.json())
.then(students => console.log(students.length))
.catch (err => console.log(err)
```

Success / Failure callbacks:

```
// definition
const processEven = i => new Promise( (resolve, reject) => {
    if (i % 2 === 0) {
        resolve(i);
    } else {
        reject(i);
    }
});

// use

processEven(4)
.then ( it => {console.log(it); return it} ) // auto promotion
.then ( it => processEven(it+1))
.catch( err => console.log( "Error: " + err))
```

Async/ Await

```
const foo = async i => {
    const x = await processEven(i).catch( err => err);
    console.log("foo: " + x);
};
foo(4);
```

Other variant:

```
async function foo(i) {
    try {
        const x = await processEven(i);
        console.log("foo: " + x);
    }
    catch(err) { console.log(err); }
};
foo(4);
```

DataFlow

Coordination schemata

> Similar to concurrency

1. No coordination needed
2. Sequence (of side effects)
3. Dependency on former results

No Coordination

> Nothing to do !

- Execution model: confident
- All actions run independently

Sequence

> Actor

- In a sequence of actions, each action can only start if the preceding one has finished
- How to achieve this => Delegated Coordination => Scheduler

Result Dependency

- Action B and C need the result of action A
- A must be executed **exactly once** before B and C
- How to do this => Implicit Coordination => DataFlowVariable

Scheduler Idea

- Queue (FIFO) of functions that are started with a lock
- Callback unlocks

DataFlowVariable

- Function, that sets a value if it is not already set. Returns the value.
- Lazy: Access to variables that will become available later
- Trick: Do not set the value, but a function that returns the value

Modules

Why Modules?

- Organize Code
- Clear Dependencies
- Avoid Errors: Globals, Scoping, Namespaces

```
// avoid something like this in your html document
<script src="fileA.js">
<script src="fileB.js">
<script src="fileC.js">
// if fileA.js has a reference on fileC.js it won't work !!!
```

Distinguish

- How I want to edit the code
- How I want to deliver the code

ES6 Modules are not

✕ Packages (those have versions)

✕ Dependencies, Libraries, Releases

✕ Units of publication

✕ Objects

Package Manager

> Installier mir mal das Packet mit Version xy

`webpack`, `npm`, `yarn`, ...

Build Tools

> Build automatisierung

`webpack`, `npm`, `grunt`, `gulp`, ...

Legacy module systems

> Bevor ES6 Module System angeboten hat

`CommonJS`, `AMD`, `UMD`, ...

Legacy Module Loader / Bundler

`RequireJS`, `SystemJS`, `browserify`, ...

Modules are async

```
// Use URI format as follows: "./myFile.js"
<script src="./myFile.js" type="module"> // type implies "defer"
import ("./myFile.js").then( modules => ... )
// invasive, transitive impact
```

Import Variants

```
// most used
import "module-name";
import { export1, export2 } from "module-name";

// other variants
import defaultExport from "module-name";
import * as name from "module-name";
import { export } from "module-name";
import { export as alias } from "module-name";
var promise = import("module-name");
```

Export Variants

```
// most used
export { name1, name2, ... , nameN };

// other variants
export function FunctionName() { .. }
export const name1, name2, ... , nameN; // or let
export class ClassName { .. }
export default expression;
export { name1 as default, .. };
export * from .. ;
export { name1, name2, ... , nameN } from .. ;
```