

FUNKTIONALE PROGRAMMIERUNG MIT JAVASCRIPT

λ

1. April:	Michael Häuslmann
15. April:	Marinus Noichl

QUELLEN & SOURCE- UND BEISPIELCODE

GITHUB REPOSITORY

<https://github.com/mihaeu/afp-js>

build passing license MIT node 5.9.1



AGENDA

- ☐ Geschichte & Herkunft
- ☐ Sprachgrundlagen & Eigenschaften
- ☐ Funktionale Konzepte
- ☐ Anwendungsbereiche
- ☐ Beispiele

Dauer: ~45min (exkl. Fragen)

GESCHICHTE & HERKUNFT

Wer ist an der Hochschule bereits mit JavaScript in Berührung gekommen?

Wer arbeitet privat/beruflich mit JavaScript?

Wer meint JavaScript ist eine „schöne“ Sprache?

GESCHICHTE & HERKUNFT

[...] you have the power to define your own subset. You can write better programs by relying exclusively on the good parts.

- Douglas Crockford, JavaScript - The Good Parts

GESCHICHTE & HERKUNFT

- 1995 von Netscape entwickelt (in 10 Tagen) für deren Browser
- verschiedene Implementierungen u.a. von Microsoft (JScript)
- Sprachstandard erst später entstanden (ECMAScript)
- aktueller Standard ist ES6 und wird von den meisten modernen Browsern zum Teil implementiert
- viele Runtime Engines, Cross-Compiler und Super-Sets (GWT, TypeScript, Dart, Coffeescript, ClojureScript...)



```
import com.google.gwt.core.client.EntryPoint;  
import com.google.gwt.user.client.Window;  
  
public class HelloWorld implements EntryPoint {  
    public void onModuleLoad() {  
        Window.alert("Hello, World!");  
    }  
}
```




Python + Ruby + Haskell = ❤️

```
# Conditions:  
number = -42 if opposite  
  
# Functions:  
square = (x) -> x * x  
  
# Splats:  
race = (winner, runners...) ->  
print winner, runners  
  
# Array comprehensions:  
cubes = (math.cube num for num in list)
```


TypeScript

```
class Greeter {  
    constructor(public greeting: string) { }  
  
    greet() {  
        return "<h1>" + this.greeting + "</h1>";  
    }  
};  
  
var greeter = new Greeter("Hello, world!");  
  
document.body.innerHTML = greeter.greet();
```

GESCHICHTE & HERKUNFT

- Keine (ernsthaften) Alternativen
- früher ausschließlich im Frontend, jetzt überall (Backend, Desktop- und Mobileapps)
-  als Java Umgebung fürs Backend (mit V8 von Google als Runtime)
- unsere Meinung: sehr viel Hype, aber auch echte Chancen

AGENDA

- ✓ *Geschichte & Herkunft*
 - *kommt aus dem Web*
 - *jetzt überall*
 - *viele Altlasten*
 - *sehr aktive und beliebt*
- **Sprachgrundlagen & Eigenschaften**
- Funktionale Konzepte
- Anwendungsbereiche
- Beispiele

SPRACHGRUNDLAGEN & EIGENSCHAFTEN

```
var schlafEin = (anzahlSchafe) => {  
  var i = 0, ausgabe = [];  
  for (i = anzahlSchafe; i > 0; --i) {  
    if (i === 1) {  
      ausgabe.push(1 + " Schaf");  
    } else {  
      ausgabe.push(i + " Schafe");  
    }  
  }  
  return ausgabe.join("\n") + "\nZzzz ...";  
};  
schlafEin(5);
```

SPRACHGRUNDLAGEN & EIGENSCHAFTEN DATENSTRUKTUREN

```
// boolean  
true  
false  
  
// number  
1  
3.1415  
(3.1415).toString()  
  
// string  
'Hello'  
  
// regex  
/java[sS]cript/
```

Alles ist ein Objekt (auch Funktionen selbst)!

SPRACHGRUNDLAGEN & EIGENSCHAFTEN DATENSTRUKTUREN

```
// assoziatives Array (functioncal scope)
var arr = [1, 2, 3];

// Objekte ähnlich wie JSON (global scope)
obj = {
  bezeichner: 'wert'
}

// seit ES6: Map und Set
// immutable Map (block scope)
const map = new Map([[ 1, 'one' ]]);

// Set (block scope)
let set = new Set([1, 1, 1, 2]); // Set { 1, 2 }
```

SPRACHGRUNDLAGEN & EIGENSCHAFTEN

- Interpretiert
- Schwache dynamische Typisierung
- Type Coercion
- Lexikalisches Scoping auf **Funktionsebene**
- Prototyp orientiert

SCHWACHE DYNAMISCHE TYPISIERUNG

```
// praktisch: kein List<Integer> list = new ArrayList<Integer>();  
let list = [1, 2, 3, 4, 5];
```

```
// dynamisch typisiert, keine Initialisierung notwendig  
let organization = 'Microsoft';
```

```
// ⚡ neue Variable durch Schreibfehler ⚡  
organisation = 'Google';
```

```
// schwach typisiert
```

```
let x = 1;
```

```
x = true;
```

```
x = 'true';
```

```
// number
```

```
// boolean
```

```
// string
```

⚡ Abgesehen für kleine Skripte fast nur Nachteile ⚡

TYPE COERCION

```
0 == ''           // true
0 == '0'          // true
'' == '0'         // false

false == 'false'  // false
false == '0'      // true
"\t\r\n" == 0     // true
```

⚡ Typumwandlungen z.B. bei == != + ... ⚡

♥ immer === verwenden! ♥

LEXIKALISCHES SCOPING AUF FUNKTIONSEBENE

```
var x = 3;
function func(randomize) {
  var x;                                // geht nur da functional scope
  if (randomize) {                       // bei let error
    let x = Math.random();
    return x;
  }
  return x;
}
func(false);                            // undefined
```

♥ immer let verwenden! ♥

PROTOTYP ORIENTIERT

- Keine Klassen, Methoden, Konstruktoren, Module (zumindest vor ES6)
- Aber alles über Prototypen möglich

```
var Animal = (function() {  
    function Animal(name) {  
        this.name = name;  
    }  
  
    Animal.prototype.move = function(meters) {  
        return this.name + " moves " + meters + "m.";  
    };  
  
    return Animal;  
})();
```

PROTOTYP ORIENTIERT

Inheritance :(

```
function Snake(name, isPoisonous) {  
  Animal.call(this, name); // super(name)  
  this.isPoisonous = isPoisonous;  
}  
  
Snake.prototype = Object.create(Snake.prototype);  
Snake.prototype.constructor = Snake;  
Snake.prototype.move = function (meters) {  
  return this.name + " wiggles " + meters + "m.";  
};
```

PROTOTYP ORIENTIERT

(UNDER THE HOOD)

Seit ES6 viel syntaktischer Zucker:

```
class Animal {  
  constructor(name) {  
    this.name = name;  
  }  
  move(meters) {  
    return this.name + " moves " + meters + "m.";  
  }  
}  
  
class Snake extends Animal {  
  move(meters) {  
    return this.name + " wiggles " + meters + "m."  
  }  
}
```

AGENDA

- ☒ Geschichte & Herkunft
- ☒ *Sprachgrundlagen & Eigenschaften*
 - *interpretiert & schwach dynamisch typisiert*
 - *Prototyp-orientiert*
 - *viele Fallen*
 - *knappe Schreibweise*
- ☐ **Funktionale Konzepte**
- ☐ Anwendungsbereiche
- ☐ Beispiele

FUNKTIONALE KONZEPTE

THE GOOD

- Funktionen waren schon immer ein first-class-citizen
- seit ES6 Tail Recursion!
- eingebaute Funktionen höherer Ordnung: filter, reduce, map (aber teils untypische Implementierungen)

THE BAD

- keine Lazy Evalution (aber es gibt Libraries)
- viele Seiteneffekte
- kein Currying, Pattern Matching, ...

HÖHERE FUNKTIONEN IN JAVASCRIPT: FILTER()

Welcher Ansatz ist einfacher zu verstehen und hat weniger mögliche Fehlerquellen?

```
// imperative
var data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
for (var i = 0, result = []; i < data.length; i++) {
    if (data[i] % 2 === 0) {
        result.push(data[i]);
    }
}
return result;
```

```
// functional
return [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].filter(i => i % 2 === 0);
```


HÖHERE FUNKTIONEN IN JAVASCRIPT: MAP()

Was mache ich, wenn ich eine ähnliche Funktion wie z.B. die Quadratwurzel brauche?

```
// imperative
data = [0, 1, 2, 3];
for (var i = 0, result = []; i < data.length; i++) {
  result.push(data[i] * data[i]);
}
return result;
```

```
// functional
return [0, 1, 2, 3].map(i => i * i);
```

HÖHERE FUNKTIONEN IN JAVASCRIPT: REDUCE()

```
// imperative
let sum = 0;
for (const i of [1, 2, 3, 4]) {    // neues immutable for seit ES6
    sum += i;
}
return sum;

// functional
return [1, 2, 3, 4].reduce((i, j) => i + j);
```

AGENDA

- ☒ Geschichte & Herkunft
- ☒ Sprachgrundlagen & Eigenschaften
- ☒ *Funktionale Konzepte*
 - *funktionale Programmierung möglich*
 - *viel Handarbeit oder Erweiterungen nötig*
 - *Performanz an manchen Stellen problematisch*
- ☐ **Anwendungsbereiche**
- ☐ Beispiele

ANWENDUNGSBEREICHE

- Web Frontend, Server Backend, Mobile- & Desktopapps: Templating, APIs, DOM, DB Operationen, ...
- viele Sachen laufen nebenbei
- JavaScript Umgebungen arbeiten aber mit einem einzigen Thread
- daher keine Möglichkeit zur Parallelisierung
- aber JavaScript Umfeld (Web) und Node Architektur Event-basiert

ANWENDUNGSBEREICHE

KURZER AUSFLUG: NODE ARCHITEKTUR

THE COST OF IO

L1-Cache	3 cycles
L2-Cache	14 cycles
RAM	250 cycles
Disk	41 000 000 cycles
Network	240 000 000 cycles

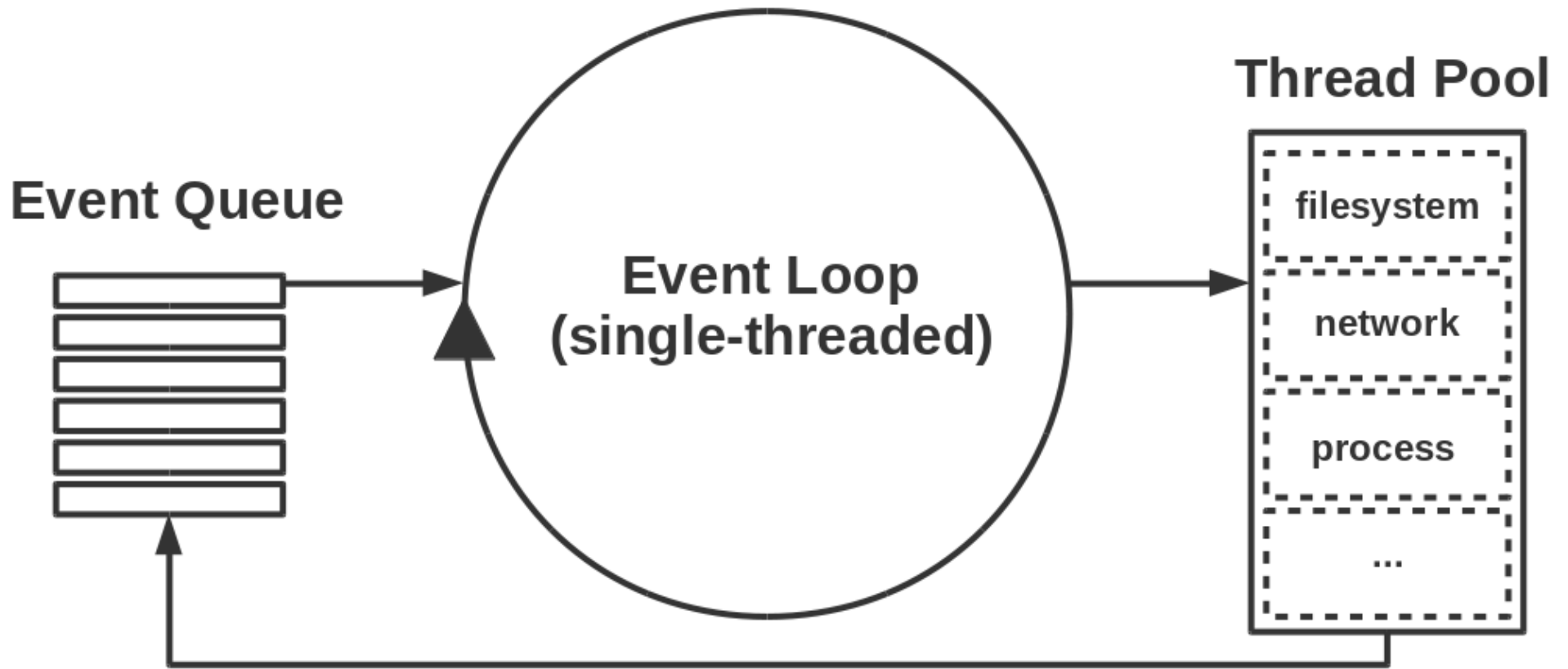
ANWENDUNGSBEREICHE:

EVENT DRIVEN PROGRAMMING

Idee: Langsame externe Events asynchron verarbeiten und weitermachen bis das Ergebnis kommt

Ergebnis: Je nach Anwendungsbereich sehr hoher Durchsatz

EVENT LOOP



ANWENDUNGSBEREICHE:

EVENT DRIVEN PROGRAMMING

```
fs.readFile('config.js',
  // some time passes...
  function(error, buffer) {
    // the result now pops into existence
    http.get(options, function(resp){
      resp.on('data', function(chunk){
        //do something with chunk
      });
    }).on("error", function(e){
      console.log("Got error: " + e.message);
    });
  }
);
```

⚡ Callback Hell ⚡ Mehr imperativ als deklarativ ⚡

ANWENDUNGSBEREICHE:

EVENT DRIVEN PROGRAMMING

Lösung: Futures/Promises (Continuation Monad)

```
fs.promisifiedReadFile('config.js')  
  .then(fetchSomethingFromWeb)  
  .then(processThatData)  
  .then(saveItToTheDatabase)  
  .catch(function(error) { console.log(error); });
```

ANWENDUNGSBEREICHE:

EVENT DRIVEN PROGRAMMING MIT PROMISES

Viele Implementierungen auch mit, fold, forEach, map etc.

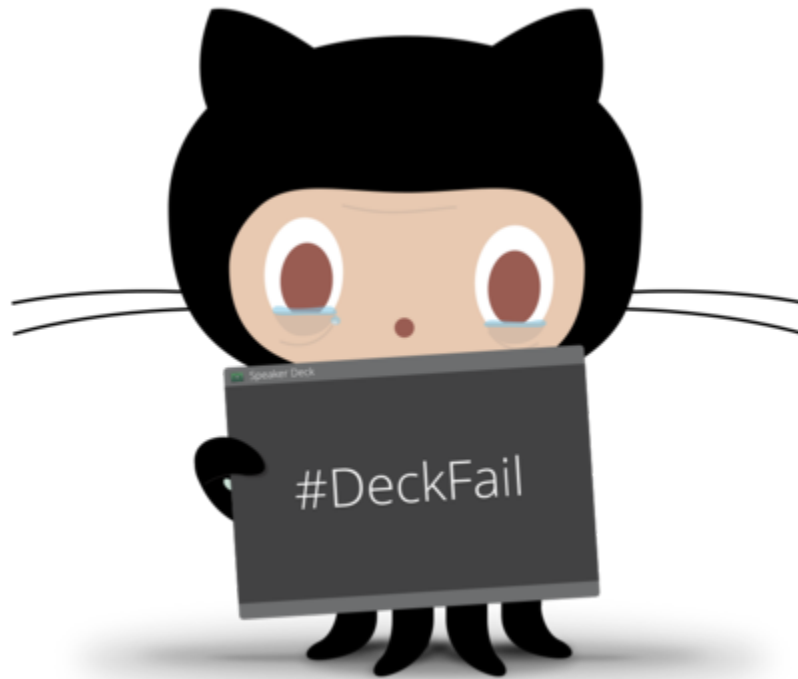
```
fs.promisifiedReadDir("/home/user/workspace")  
  .map(fs.promisifiedReadFile)  
  .reduce((total, content) => total += content.length, 0)  
  .then(result => console.log(result))  
  .catch(error => console.log(error));
```

AGENDA

- ☒ Geschichte & Herkunft
- ☒ Sprachgrundlagen & Eigenschaften
- ☒ Funktionale Konzepte
- ☒ *Anwendungsbereiche*
 - *Single-Thread Architektur problematisch für komplexe Berechnungen*
 - *Event-basierte Programmierung gut mit funktionaler Programmierung kombinierbar*
- ☐ **Beispiele**

BEISPIELE: LIVE CODING

- Sort
- Currying
- Funktionen höherer Ordnung



ZUSAMMENFASSUNG

- ✓ *Geschichte & Herkunft*
- ✓ *Sprachgrundlagen & Eigenschaften*
- ✓ *Funktionale Konzepte*
- ✓ *Anwendungsbereiche*
- ✓ *Beispiele*

ZUSAMMENFASSUNG

JavaScript ist ...

- ... einfach zu lernen
- ... überall zu verwenden
- ... mit vielen Altlasten und Problemen
- ... für funktionale Programmierung geeignet
- ... ist aber nicht für rein funktionale Programmierung entworfen

QUELLEN

BÜCHER

- [JavaScript - The Good Parts](#) von Douglas Crockford
- [Exploring ES6](#) von Dr. Axel Rauschmayer
- [Das Curry-Buch - Funktional programmieren lernen mit JavaScript](#) von Stefanie Schirmer, Hannes Mehnert, Jens Ohlig

BLOGS

- [Ode to Code](#)
- [@ality – JavaScript and more](#)

SONSTIGE QUELLEN & INTERESSANTE LINKS

- [Cost of IO](#)
- [Callbacks are imperative, Promises are functional](#)
- [Monads in JavaScript](#)
- [Promises are the monad of asynchronous programming](#)
- [JavaScript the Good Parts \(Online Video Kurs\)](#)
- [What is the appeal of dynamically-typed languages?](#)
- [ClojureScript](#)


```
let add = function(a, b) {  
  return a + b;  
}  
let add2 = (a, b) => a + b;  
  
let map = (fn, xs) => {  
  if (!xs.length) return [];  
  return [fn(xs[0])].concat(map(fn, xs.slice(1)));  
};
```

```
let inc = a => a + 1;  
map(inc, [0, 1, 2]);
```

```
let applyFn = (fn, x) => (y) => fn(x, y);  
let inc2 = applyFn(add, 1);
```

```
let curry = (fn, ...args) => fn.length === args.length  
  ? fn(...args)  
  : curry.bind(this, fn, ...args);  
let inc3 = curry(add)(1);
```

```
let sort = xs => {  
  if (xs.length === 0) return [];  
  let pivot = xs[0], t = xs.slice(1);  
  return sort(t.filter(x => x < pivot))  
    .concat(pivot)  
    .concat(sort(t.filter(x => x >= pivot)));  
}
```