WBE: JAVASCRIPT

FUNKTIONEN

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

FUNKTION DEFINIEREN

```
// Zuweisung einer anonymen Funktion
const squareV1 = function (x) {
  return x * x
}

// Funktionsdeklaration
function squareV2 (x) {
  return x * x
}

// Pfeilnotation
const squareV3 = x => x * x
```

FUNKTIONSDEKLARATION

```
1 console.log(square(5))
2
3 function square (x) {
4   return x * x
5 }
```

- Alternative Möglichkeit, Funktionen einzuführen
- Funktionsdeklarationen werden zuerst ausgewertet
- Aufruf kann daher in Script vor Deklaration stehen

FUNKTION IN PFEILNOTATION

```
/* Pfeilnotation mit Ausdruck */
const square = x => x * x

/* Pfeilnotation mit Block */
const add = (x, y) => {
  return x + y
}

/* Pfeilnotation mit leerer Parameterliste */
const randomize = () => {
  return Math.random()
}
```


FUNKTIONEN SIND OBJEKTE

```
1 let launchMissiles = function () {
2   missileSystem.launch("now")
3 }
4 if (safeMode) {
5   launchMissiles = function () {/* do nothing */}
6 }
```

- Funktionen sind spezielle, aufrufbare Objekte
- Sie können an Variablen gebunden werden
- Diese Bindung kann jederzeit neu zugewiesen werden (ausser es ist eine Konstante)
- Funktionen auch als Parameter und Rückgabewert möglich: First Class Citizens

FUNKTIONEN SIND OBJEKTE

- Man kann jederzeit Attribute oder Methoden hinzufügen
- Sie haben bereits vordefinierte Methoden

```
> const add = (x, y) => x + y
> add.doc = "This function adds two values"

> add(3,4)
7

> [add.toString(), add.doc]
[ '(x, y) => x + y', 'This function adds two values' ]
```

REKURSIVE FUNKTIONEN

```
const factorial = function (n) {
  if (n <= 1) {
    return 1
  } else {
    return n * factorial(n-1)
  }
}

/* oder kurz: */
const factorial = n => (n<=1) ? 1 : n * factorial(n-1)
console.log(factorial(10))     // → 3628800
```

- Funktionen können sich selbst aufrufen
- Abwägen zwischen eleganter und performanter Lösung

GÜLTIGKEITSBEREICH (SCOPE)

```
let   m = 10       // variable with block scope
const n = 10       // constant with block scope
var   p = 10       // variable with function scope
```

- Am besten: `const`, wenn veränderlich: `let`
- Funktions-Scope (`var`) kann verwirren

GÜLTIGKEITSBEREICH (SCOPE)

```
const demo = function () {
  let x = 10
  if (true) {
    let y = 20
    var z = 30
    console.log(x + y + z)
    /* → 60 */
  }
  /* y is not visible here */
  console.log(x + z)
  /* → 40 */
}
```


GLOBALE VARIABLEN

- Ausserhalb von Funktionen definiert
- Oder in Funktionen, aber `const`, `let` oder `var` vergessen
- Gültigkeitsbereich möglichst einschränken (Block, Funktion)

```
const add = function (a, b) {
  result = a + b        /* schlecht! - strict mode: Fehler! */
  return result
}

console.log(add(3,4))   /* → 7       */
console.log(result)     /* → 7 (!)   */
```

INNERE FUNKTIONEN

```
const hummus = function (factor) {

  const ingredient = function (amount, unit, name) {
    let ingredientAmount = amount * factor
    if (ingredientAmount > 1) {
      unit += "s"
    }
    console.log(`${ingredientAmount} ${unit} ${name}`)
  }

  ingredient(1, "can", "chickpeas")
  ingredient(0.25, "cup", "tahini")
  // ...
}
```


LEXICAL SCOPING

- Die Menge der sichtbaren Variablenbindungen wird bestimmt durch den Ort im Programmtext
- Jeder lokale Gültigkeitsbereich sieht alle Gültigkeitsbereiche, die ihn enthalten
- Alle Gültigkeitsbereiche sehen den globalen Gültigkeitsbereich

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

PARAMETER VON FUNKTIONEN

```
function square (x) { return x * x }
console.log(square(4, true, "hedgehog"))   // → 16
```

- Anzahl Parameter muss nicht mit der Anzahl beim Aufruf übergebener Argumente übereinstimmen
- Fehlende Argumente: sind `undefined`
- Überzählige Argumente: werden ignoriert

PARAMETER VON FUNKTIONEN

- Überladen wie in Java somit nicht möglich
- Ähnlicher Effekt durch Test auf `undefined`

```
function minus (a, b) {
  if (b === undefined) return -a
  else return a - b
}

console.log(minus(10))       /* → -10   */
console.log(minus(10, 5))    /* → 5     */
```


DEFAULT-PARAMETER

```
function power (base, exponent=2) {
  let result = 1
  for (let count = 0; count < exponent; count++) {
    result *= base
  }
  return result
}

console.log(power(4))        /* → 16   */
console.log(power(2, 6))     /* → 64   */
```

- Falls Argument `undefined` ist, wird Default eingesetzt
- Default-Parameter stehen am Ende der Parameterliste

REST-PARAMETER

- Übergebene Argumente in Array verfügbar
- Vorher können normale Parameter stehen

```
function max (...numbers) {
  let result = -Infinity
  for (let number of numbers) {
    if (number > result) result = number
  }
  return result
}

console.log(max(4, 1, 9, -2))     /* → 9 */
```


arguments

- Rest-Parameter wurde mit ES2015 eingeführt
- Alternative: `arguments`
- Das ist ein Array-ähnliches Objekt

```
function f () {
  return arguments.length
}
function g (...args) {
  return args.length
}

console.log(f(1,2,3,4))           /* → 4   */
console.log(g("hello", "world"))  /* → 2   */
```

SPREAD -SYNTAX

- Spread-Operator `...`
- Fügt den Array-Inhalt in die Parameterliste ein
- Analog Spread-Syntax in Arrays

```
let numbers = [5, 1, 7]
console.log(max(...numbers))          /* → 7 */
console.log(max(9, ...numbers, 2))    /* → 9 */

// zum Vergleich: Array
let more = ["a", "b"]
numbers = [5, 1, ...more, 7]
console.log(numbers)                  /* → [ 5, 1, 'a', 'b', 7 ] */
```


ARRAYS DESTRUKTURIEREN

- Wie bei der Zuweisung können Arrays auch bei der Parameterübergabe destrukturiert werden
- Vermeidet das spätere Zugreifen über den Array-Index

```
const func = ( [a, b] ) => `${a}.${b}`
let result = func([5, 1])            /* → '5.1' */

// zum Vergleich: return-Wert destrukturieren
function gethttp (url) {
  if (...) return [404, "not found"]
}
let [code, message] = gethttp(url)
```


OBJEKTE DESTRUKTURIEREN

```
let bar = 87
let obj = { foo: 12, bar, baz: 43 }

const selectFoo = ( {foo} ) => foo
console.log(selectFoo(obj))          /* → 12 */
```

- Nur einzelne Attribute aus einem (möglicherweise sehr grossen) Objekt übernehmen
- Vermeidet das spätere Zugreifen über den Attributnamen

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

ABSTRAKTION

- Der richtige Grad an Abstraktion
  - macht Programme lesbarer und verständlicher
  - reduziert Fehler
- Funktionen als wichtiges Abstraktionsmittel

```
/* Summe der Zahlen von 1 bis 10 */
let total = 0, count = 1
while (count <= 10) {
  total += count
  count += 1
}
let result = total

/* mit Abstraktionen sum und range wenn verfügbar */
let result = sum(range(1, 10))
```


FUNKTIONEN HÖHERER ORDNUNG

- Funktionen, welche Funktionen als Parameter oder Rückgabewert haben
- Sie bieten weitere Abstraktionsmöglichkeiten

```
function repeat (n, action) {
  for (let i = 0; i < n; i++) {
    action(i)
  }
}

let labels = []
repeat(4, i => {
  labels.push(`Unit ${i + 1}`)
})
console.log(labels)   // → ["Unit 1", "Unit 2", "Unit 3", "Unit 4"]
```


ARRAY VERARBEITEN

```
for (let item of [1,2,3]) {
  console.log(item)
}
/* → 1  → 2  → 3 */

[1,2,3].forEach(item => console.log(item))
/* → 1  → 2  → 3 */
```

- `for..of` als Variante zur normalen for-Schleife
- `forEach` ist eine Methode von Arrays, welche eine Funktion bekommt und nur über zugewiesene Array-Stellen iteriert

ARRAY FILTERN

```
> let num = [5, 2, 9, -3, 15, 7, -5]

> num.filter(n => n>0)
[ 5, 2, 9, 15, 7 ]

> num.filter(n => n%3==0)
[ 9, -3, 15 ]
```

- Neues Array wird erstellt
- Elemente, die Prädikat erfüllen, werden übernommen

ARRAY ABBILDEN

```
> let num = [5, 2, 9, -3, 15, 7, -5]

> num.map(n => n*n)
[ 25, 4, 81, 9, 225, 49, 25 ]
```

- Funktion für jedes Element aufgerufen
- Neues Array mit den Ergebnissen wird gebildet

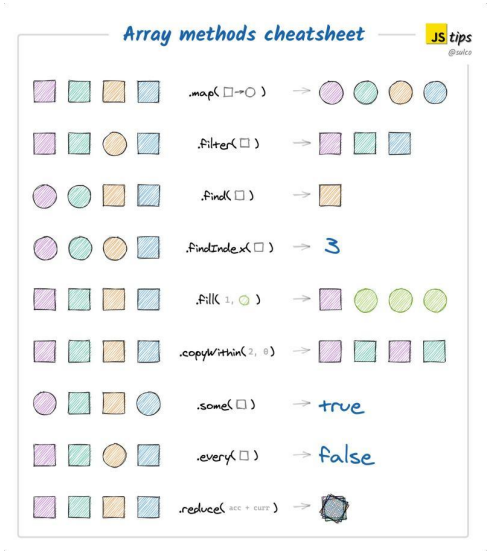
ARRAY REDUZIEREN

```
> let num = [5, 2, 9, -3, 15, 7, -5]

> num.reduce((curr, next) => curr+next)
30

> num.reduce((curr, next) => 'f('+curr+','+next+')')
'f(f(f(f(f(f(5,2),9),-3),15),7),-5)'
```

- Erste zwei Elemente mit Funktion verknüpfen
- Zwischenresultate mit jeweils nächstem Element verknüpfen
- Reduzieren der Liste auf einen Wert



Array-Methoden

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

CLOSURES

```
function wrapValue (n) {
  let local = n
  let func = () => local
  return func
}

let wrap1 = wrapValue(1)
let wrap2 = wrapValue(2)

console.log(wrap1())
console.log(wrap2())
```

- `local` steht in `func` zur Verfügung (umgebender Gültigkeitsbereich)
- Das gilt nach Beenden von `wrapValue` weiterhin
- Funktion ist in Gültigkeitsbereich eingeschlossen (enclosed)
- Ausgabe?

CLOSURES

```
function wrapValue_v1 (n) {
  let local = n
  let func = () => local
  return func
}

/* kürzer: */
function wrapValue_v2(n) {
  return () => n
}

/* noch kürzer: */
const wrapValue_v3 = (n) => () => n
```


CLOSURES: ÜBUNG

```
const prefix = (pre) => (text) => pre + text
```

- Überlegen Sie, was die Funktion `prefix` macht
- Geben Sie ein Beispiel an, wie sie eingesetzt werden kann

FUNKTIONEN DEKORIEREN

```
function trace (func) {
  return (...args) => {
    console.log(args)
    return func(...args)
  }
}

/* Fakultätsfunktion */
let factorial = (n) => (n<=1) ? 1 : n * factorial(n-1)

/* Tracer an Funktion anbringen */
factorial = trace(factorial)

/* Aufruf */
console.log(factorial(3))    // → [ 3 ] → [ 2 ] → [ 1 ] → 6
```

PURE FUNKTIONEN

- Rückgabewert der Funktion ist ausschliesslich abhängig von den übergebenen Argumenten
- Keine weiteren Abhängigkeiten
- Keine Seiteneffekte

Pure Funktionen haben zahlreiche Vorteile. Sie sind gut kombinierbar, gut zu testen, problemlos in verschiedenen Programmen einsetzbar.

Funktionen mit Seiteneffekten sind manchmal nötig. Wenn möglich sollten aber pure Funktionen geschrieben werden.

FUNKTIONALES PROGRAMMIEREN

- Variante von `reduce`, die eine Funktion zurückgibt
- Currying: Umwandeln in Funktionen mit einem Argument
- Nun lässt sich die Summe eines Arrays elegant definieren

```
1 const reduce = f => init => array => array.reduce(f, init)
2 const add = (m, n) => m + n
3
4 reduce (add) (0) ([1,2,3,4])            /* → 10 */
5
6 /* Summe der Elemente eines Arrays */
7 const sum = reduce(add)(0)
8
9 sum([1,2,3,4])                          /* → 10 */
```


FUNKTIONALES PROGRAMMIEREN

- Funktionen sind in JavaScript ausserordentlich mächtig und sehr flexibel einsetzbar
- JavaScript unterstützt funktionales Programmieren, ist aber eine Multiparadigmensprache
- Beispiel für eine rein funktionale Sprache: Haskell

Das funktionale Paradigma erlaubt elegante Lösungen zu vielen Problemen und wird von Programmiersprachen zunehmend unterstützt

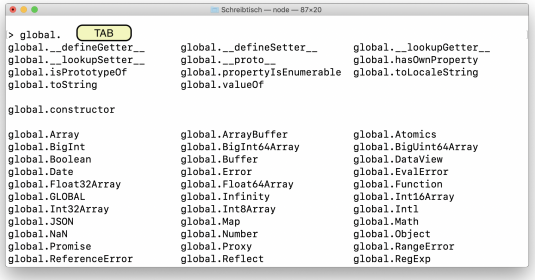
In WBE kann es aber nicht weiter vertieft werden

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures und funktionale Programmierung
- Mehr zu Node.js

NODE.JS KONSOLE

- Auf der Konsole können Node.js-Objekte untersucht werden
- Ausgabe aller Attribute und Methoden



KOMMANDOZEILENARGUMENTE

- Über `process.argv` zugreifbar
- Array von Kommandozeilenargumenten als Strings

```
/* args.js */
process.argv.forEach((val, index) => {
  console.log(`${index}: ${val}`)
})
```

```
$ node args.js eins 2 iii
0: /opt/local/bin/node
1: /Users/guest/Desktop/args.js
2: eins
3: 2
4: iii
```


EINGABEN VON DER KOMMANDOZEILE

```
1 const readline = require('readline').createInterface({
2   input: process.stdin,
3   output: process.stdout
4 })
5
6 readline.question(`What's your name?`, name => {
7   console.log(`Hi ${name}!`)
8   readline.close()
9 })
```

https://nodejs.org/api/readline.html

MODULE IN JAVASCRIPT

- Einfachste Variante: alles in Funktion einpacken
- Immediately Invoked Function Expressions
- Globaler Namensraum nicht beeinflusst
- Diverse Varianten, z.B. Rückgabe globaler Elemente

```
(function () {
  let foo = function () {...}
  let bar = 'Hello world'
  console.log(bar)
})()
```

MODULSYSTEM (COMMONJS)

```
/* car-lib.js */
const car = {
  brand: 'Ford',
  model: 'Fiesta'
}

module.exports = car
```

```
/* other js file */
const car = require('./car-lib')
```


MODULSYSTEM (ES6)

```
/* square.js */
const name = 'square'
function draw (ctx, length, x, y, color) { ... }
function reportArea () { ... }

export { name, draw, reportArea }
```

```
/* other js file */
import { name, draw, reportArea } from './modules/square.js'
```


NPM

- Online Repository von JavaScript-Modulen
- Paketverwaltung für Node.js
- Pakete werden im Verzeichnis `node_modules` installiert
- Dabei werden auch Abhängigkeiten aufgelöst
- Beispiel:

```
$ # lokale Installation im Projekt, Verzeichnis node_modules
$ npm install lodash

$ # globale Installation eines Pakets
$ npm install -g cowsay
```


NPM

- Projektdatei `package.json` mit Liste benötigter Pakete
- Ausserdem weitere Projektinformationen

```
$ # Pakete wie in package.json beschrieben installieren
$ npm install

$ # Paket installieren und in package.json eintragen
$ npm install --save lodash

$ # Paket installieren und in package.json unter devDependencies eintragen
$ npm install --save-dev lodash
```

NPM

```
$ # installierte Pakete ausgeben
$ npm list

$ # nur oberste Ebene (ohne abhängige Pakete)
$ npm list --depth=0

$ # Version eines Pakets / alle verfügbaren Versionen
$ npm list <package>
$ npm view <package> versions

$ # Installation einer bestimmten Version
$ npm install cowsay@1.2.0

$ # Paket entfernen (-S: auch in package.json)
$ npm uninstall <package>
```


NPX

- Code von Node-Paketen starten
- Paket muss dazu nicht erst installiert werden
- Seit npm 5.2 enthalten, kann auch separat installiert werden
- Beispiel: React App anlegen ohne Tool `create-react-app` vorher zu installieren:

```
$ npx create-react-app my-react-app
```


WEITERE TOOLS

- Vite: Entwicklungsprozess automatisieren
  https://vitejs.dev
- Webpack: Module Bundler, Abhängigkeiten auflösen
  https://webpack.js.org
- Yarn: Paketverwaltung, Alternative zu npm
  https://yarnpkg.com

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 3, 5 und 10 von:
  Marijn Haverbeke: Eloquent JavaScript
  https://eloquentjavascript.net/