

WBE: JAVASCRIPT FUNKTIONEN



ÜBERSICHT

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

ÜBERSICHT

- Funktionen definieren
- Parameter von Funktionen
- Funktionen höherer Ordnung
- Closures
- 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
```

Hier noch einmal als Wiederholung einige Hinweise zur Definition von Funktionen:

Eine anonyme Funktion kann einem Namen zugewiesen werden. Funktionen sind spezielle Objekte, die ausführbar sind. Im Beispiel wird die Funktion dem Namen `squareV1` als Konstante zugewiesen. Hätten wir eine Variable verwendet, könnten wir später einen neuen Wert (Funktion oder etwas anderes) dem Namen `squareV1` zuweisen.

Die Funktion `squareV1` kann erst nach der Zuweisung aufgerufen werden. Das ist anders bei der Funktionsdeklaration. Diese wird bereits beim Laden des Scripts ausgewertet. Das bedeutet, dass `squareV2` bereits weiter oben im Script aufgerufen werden könnte. Ob und wann das sinnvoll ist, ist eine andere Frage.

Beim Zuweisen der Funktion in Pfeilnotation wird ebenfalls eine anonyme Funktion zugewiesen. Funktionen in Pfeilnotation unterscheiden sich etwas von mit `function` eingeführten Funktionen. Mehr dazu später.

FUNKTION DEFINIEREN

```
/* 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()  
}
```

Speaker notes

- Funktionen können auch eine leere Parameterliste haben.
- Bei Funktionen in Pfeilnotation, die genau einen Parameter haben, muss die Parameterliste nicht geklammert werden
- Bei Funktionen in Pfeilnotation, die nur einen Ausdruck berechnen, können die Klammern um den Funktionskörper und die `return`-Anweisung fehlen
- Funktionen mit Anweisungsblock geben `undefined` zurück, wenn die `return`-Anweisung fehlt

FUNKTIONEN SIND OBJEKTE

- Funktionen sind spezielle, aufrufbare 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' ]
```


Speaker notes

Weitere vordefinierte Methoden von Funktionen sind `call`, `apply` und `bind`.
Mehr dazu in einer späteren Lektion.

REKURSIVE FUNKTIONEN

```
1  const factorial = function (n) {  
2    if (n <= 1) {  
3      return 1  
4    } else {  
5      return n * factorial(n-1)  
6    }  
7  }  
8  
9  /* oder kurz: */  
10 const factorial = (n) => (n<=1) ? 1 : n * factorial(n-1)  
11  
12 /* Aufruf: */  
13 console.log(factorial(10))      // → 3628800
```

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)

```
1  const demo = function () {  
2    let x = 10  
3    if (true) {  
4      let y = 20  
5      var z = 30  
6      console.log(x + y + z)  
7      /* → 60 */  
8    }  
9    /* y is not visible here */  
10   console.log(x + z)  
11   /* → 40 */  
12 }
```

Speaker notes

Am besten ist es, `var` zu vermeiden. Mit `var` eingeführte Variablen können sogar erneut deklariert werden:

```
function test () {  
  var i=10  
  var i=11  
  console.log(i)  
}
```

Das gilt nicht für `let`. Die folgende Funktion produziert einen Syntax Error, was das eigentlich gewünschte Verhalten ist:

```
function test () {  
  let i=10  
  let i=11  
  console.log(i)  
}
```

Da Variablen mit `var` redeclariert werden können, können auch mehrere Schleifen die gleiche Laufvariable enthalten. Diese ist aber auch ausserhalb der Schleife noch zugreifbar. Daher auch hier besser `let` verwenden:

```
function loop (n) {  
  for (var i=0; i < n; i+=1) {  
    console.log(i)  
  }  
  for (var i=0; i < n; i+=1) {  
    console.log(i)  
  }  
  console.log(i)  
}
```

Globale Variablen

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

```
1 const add = function (a, b) {  
2   result = a + b          /* schlecht! */  
3   return result  
4 }  
5  
6 console.log(add(3,4))      /* → 7      */  
7 console.log(result)       /* → 7 (!)   */
```

Speaker notes

Globale Variablen sollte man vermeiden. Tatsächlich sind globale Variablen Attribute des globalen Objekts. Im Browser ist das globale Objekt `window`, in Node.js ist es `global`:

```
> a = 17
17
> global
<ref *1> Object [global] {
  ...
  a: 17
}
```

In Modulen sind Variablen, welche ausserhalb von Funktionen definiert und nicht exportiert werden, auf das Modul beschränkt.

INNERE FUNKTIONEN

```
1  const hummus = function (factor) {  
2  
3    const ingredient = function (amount, unit, name) {  
4      let ingredientAmount = amount * factor  
5      if (ingredientAmount > 1) {  
6        unit += "s"  
7      }  
8      console.log(`${ingredientAmount} ${unit} ${name}`)  
9    }  
10  
11    ingredient(1, "can", "chickpeas")  
12    ingredient(0.25, "cup", "tahini")  
13    // ...  
14  }
```


Speaker notes

- Funktionen können **innere Funktionen** enthalten
- Die innere Funktion hat Zugriff auf die Umgebung
- Im Beispiel: sie kann auf *factor* zugreifen

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
- 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`

```
1 function minus (a, b) {  
2   if (b === undefined) return -a  
3   else return a - b  
4 }  
5  
6 console.log(minus(10))      /* → -10    */  
7 console.log(minus(10, 5))  /* → 5     */
```

DEFAULT-PARAMETER

```
1 function power (base, exponent=2) {  
2   let result = 1  
3   for (let count = 0; count < exponent; count++) {  
4     result *= base  
5   }  
6   return result  
7 }  
8  
9 console.log(power(4))           /* → 16    */  
10 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

```
1 function max (...numbers) {  
2   let result = -Infinity  
3   for (let number of numbers) {  
4     if (number > result) result = number  
5   }  
6   return result  
7 }  
8  
9 console.log(max(4, 1, 9, -2))    /* → 9 */
```

Wenn nötig kann mit einem Rest-Parameter auch eine bestimmte Anzahl Argumente erzwungen werden:

```
function createPoint (...args) {  
  if (args.length !== 2) {  
    throw new Error('Please provide exactly 2 arguments!')  
  }  
  const [x, y] = args  
  // ...  
}
```

Quelle:

https://exploringjs.com/impatient-js/ch_callables.html#parameter-handling

arguments

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

```
1 function f () {  
2   return arguments.length  
3 }  
4 function g (...args) {  
5   return args.length  
6 }  
7  
8 console.log(f(1,2,3,4))      /* → 4    */  
9 console.log(g("hello", "world")) /* → 2    */
```

Speaker notes

Die Tatsache, dass `arguments` kein echtes Array ist bedeutet, dass keine Array-Methoden wie `map` auf es anwendbar sind. Man kann aber ein echtes Array daraus machen:

```
function f(a, b) {  
  
    let normalArray = Array.prototype.slice.call(arguments)  
    // -- or --  
    let normalArray = [].slice.call(arguments)  
    // -- or --  
    let normalArray = Array.from(arguments)  
  
    let first = normalArray.shift() // OK, gives the first argument  
    let first = arguments.shift()  // ERROR (arguments is not a normal array)  
}
```

Quelle:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

Array-ähnlich heisst: es hat ein `length`-Attribut und man kann über den Index auf einzelne Attribute zugreifen.

Weiterer Unterschied: Rest-Parameter können mit normalen Parametern kombiniert werden. Dagegen enthält `arguments` immer alle Parameter.

Wichtig auch: `arguments` steht nur in mit `function` definierten Funktionen zur Verfügung, nicht aber in Funktionen, welche in Pfeilnotation definiert werden.

SPREAD -SYNTAX

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

```
1 let numbers = [5, 1, 7]
2 console.log(max(...numbers))           /* → 7 */
3 console.log(max(9, ...numbers, 2))      /* → 9 */
4
5 // zum Vergleich: Array
6 let more = ["a", "b"]
7 numbers = [5, 1, ...more, 7]
8 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

```
1 const func = ( [a, b] ) => `${a}.${b}`
2 let result = func([5, 1])           /* → '5.1' */
3
4 // zum Vergleich: return-Wert destrukturieren
5 function gethttp (url) {
6   if (...) return [400, "not found"]
7 }
8 let [code, message] = gethttp(url)
```

Speaker notes

Das zweite Beispiel zeigt, dass das Destrukturieren auch bei der Rückgabe aus Funktionen verwendet werden kann. Es ist in diesem Fall ja nichts anderes als eine Zuweisung. Auf diese Weise können Funktionen also problemlos mehrere Werte zurückgeben.

Zur Erinnerung: Arrays destrukturieren:

```
let [a, b, c, d] = [1, 2, 3]
```

```
console.log(c)
```

```
/* → 3 */
```

```
console.log(d)
```

```
/* → undefined */
```

OBJEKTE DESTRUKTURIEREN

```
1 let bar = 87
2 let obj = { foo: 12, bar, baz: 43 }
3
4 const selectFoo = ( {foo} ) => foo
5 console.log(selectFoo(obj))           /* → 12 */
```

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

OBJEKTE DESTRUKTURIEREN

Das funktioniert ganz ähnlich wie bei den Arrays. Wenn man aus einem Objekt nur bestimmte Attributwerte haben möchte, kann man diese so zuweisen.

Angenommen, wir haben ein grosses Objekt, das etwa 20 CSS-Eigenschaften beschreibt. Wenn für eine Funktion nur die Eigenschaften `lineHeight` und `fontSize` von Interesse ist, kann dies folgendermassen eingesetzt werden:

```
const css = { fontSize: 12, /* ... */, lineHeight: 1.5, /* ... */ }
```

```
function styleText (text, {lineHeight, fontSize}) {  
  /* ... */  
}
```

```
styleText(mydoc, css)
```

NAMED PARAMETERS

Beim Destrukturieren von Objekten und Arrays sind auch Default-Werte möglich. Durch Kombination der Techniken kann auch eine Art benannte Parameter umgesetzt werden:

```
function selectEntries({start=0, end=-1, step=1} = {}) {  
  return {start, end, step}  
}
```

// Aufruf:

```
selectEntries()           // → { start: 0, end: -1, step: 1 }  
selectEntries({end:10})   // → { start: 0, end: 10, step: 1 }
```

Quelle:

https://exploringjs.com/impatient-js/ch_callables.html#parameter-handling

ÜBERSICHT

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

ABSTRAKTION

- Der richtige Grad an Abstraktion
 - macht Programme lesbarer und verständlicher
 - reduziert Fehler
- Funktionen ermöglichen Abstraktion

```
1  /* Summe der Zahlen von 1 bis 10 */
2  let total = 0, count = 1
3  while (count <= 10) {
4      total += count
5      count += 1
6  }
7  let result = total
8
9  /* mit Abstraktionen sum und range */
10 let result = sum(range(1, 10))
```

Speaker notes

Die Gefahr, Fehler zu machen, ist in der Variante mit der Schleife sicher höher, wenn man annimmt, dass bewährte gut getestete Funktionen `range` und `sum` zur Verfügung stehen.

Programmieren auf einem höheren Abstraktionsniveau ist oft mit höherem Speicherverbrauch und/oder längerer Laufzeit verbunden. Wenn das aber mit geringerer Fehleranfälligkeit einhergeht, nimmt man das fast immer gerne in Kauf.

Wenn Funktionen wie `sum` und `range` bereits verfügbar sind, hat man zudem erheblich weniger Programmieraufwand. Diesen Vorteil kann man jedoch nur nutzen, wenn man diese und viele andere bereitstehenden Funktionen kennt oder zumindest bei Bedarf findet.

FUNKTIONEN HÖHERER ORDNUNG

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

```
1 function repeat (n, action) {
2   for (let i = 0; i < n; i++) {
3     action(i)
4   }
5 }
6
7 let labels = []
8 repeat(4, i => {
9   labels.push(`Unit ${i + 1}`)
10 })
11 console.log(labels)    // → ["Unit 1", "Unit 2", "Unit 3", "Unit 4"]
```

Speaker notes

Statt einer Schleife haben wir neu einen Funktionsaufruf. Die eigentliche Schleife ist in die Implementierung von `repeat` gewandert. Die Funktion `repeat` ist nur sinnvoll für Funktionen mit Seiteneffekten.

Hier noch eine weitere Funktion, welche Schleifen „weg-abstrahieren“ kann: Die Funktion `iterate` baut ein Array auf, indem jedes Element der Funktionswert des vorhergehenden ist:

```
function iterate (n, init, fn) {  
  let result = [], next = init  
  for (let i=0; i<n; i++) {  
    result.push(next)  
    next = fn(next)  
  }  
  return result  
}
```

```
iterate(5, 1, n=>n+1)      // → [ 1, 2, 3, 4, 5 ]
```

```
iterate(20, 1, n=>n*2)     // → [ 1, 2, 4, 8, 16, 32, 64, 128 ]
```

ARRAY VERARBEITEN

```
1 for (let item of [1,2,3]) {  
2   console.log(item)  
3 }  
4 /* → 1 → 2 → 3 */  
5  
6 [1,2,3].forEach(item => console.log(item))  
7 /* → 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

Die Array-Methode `forEach` überspringt also *leere* Array-Positionen:

```
> let zahlen = [1,2,,,,,7]
> zahlen.length
7
> zahlen.forEach(item => console.log(item))
1
2
7
```

Übersprungen werden aber nur nicht besetzte Stellen. Wenn etwas zugewiesen wurde, auch `undefined` oder `null`, wird die Stelle nicht mehr übersprungen:

```
> zahlen[3] = undefined
> zahlen.forEach(item => console.log(item))
1
2
undefined
7
> delete zahlen[3]
> zahlen.forEach(item => console.log(item))
1
2
7
```

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

Speaker notes

Mit *Prädikat* ist hier eine Funktion gemeint, welche einen Wahrheitswert liefert. Das kann true oder false sein oder ein Wert, der als true oder false interpretiert werden kann.

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

Speaker notes

Der eine Wert kann natürlich wieder eine Liste sein.

Als weiteres Argument kann auch noch ein Initialwert angegeben werden:

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

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

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

Wenn ein Initialwert angegeben wird, können auch leere Listen verarbeitet werden.

Neben reduce gibt es auch eine Methode reduceRight die die Liste vom Ende her verarbeitet:

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

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



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

> [1,2,3].reduce((m,n)=>m-n)
-4



> [1,2,3].reduceRight((m,n)=>m-n)
0
```

Array methods cheatsheet

JS tips
@sulco

 `.map(□ → ○)` → 


 `.filter(□)` → 

 `.find(□)` → 

 `.findIndex(□)` → **3**

 `.fill(1, ○)` → 

 `.copyWithin(2, 0)` → 

 `.some(□)` → **true**

 `.every(□)` → **false**

 `.reduce(acc + curr)` → 

Array-Methoden

ÜBERSICHT

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

CLOSURES

```
1 function wrapValue (n) {  
2   let local = n  
3   let func = () => local  
4   return func  
5 }  
6  
7 let wrap1 = wrapValue(1)  
8 let wrap2 = wrapValue(2)  
9  
10 console.log(wrap1())  
11 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?

Speaker notes

```
console.log(wrap1()) // → 1  
console.log(wrap2()) // → 2
```

Wenn eine Funktion aufgerufen wird, sieht sie die Umgebung, in der sie definiert wurde, aber nicht die Umgebung, in welcher sie aufgerufen wird.

CLOSURES

```
1  function wrapValue_v1 (n) {
2    let local = n
3    let func = () => local
4    return func
5  }
6
7  /* kürzer: */
8  function wrapValue_v2(n) {
9    return () => n
10 }
11
12 /* noch kürzer: */
13 const wrapValue_v3 = (n) => () => n
```

Speaker notes

Hier sieht man, dass die lokale Variable `local` hier gar nicht nötig ist, da der Parameter `n` ebenfalls eine lokale Bindung ist.

Die dritte Variante sieht schon fast aus wie die Funktionsdefinition in einer funktionalen Programmiersprache.

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

Speaker notes

```
// Präfix wird in Funktion addalfa gespeichert...  
let addalfa = prefix("alfa_")  
  
// ... und hier verwendet  
console.log(addalfa("starting..."))    // → 'alfa_starting...'
```

FUNKTIONEN DEKORIEREN

```
1  function trace (func) {
2    return (...args) => {
3      console.log(args)
4      return func(...args)
5    }
6  }
7
8  /* Fakultätsfunktion */
9  let factorial = (n) => (n<=1) ? 1 : n * factorial(n-1)
10
11 /* Tracer an Funktion anbringen */
12 factorial = trace(factorial)
13
14 /* Aufruf */
15 console.log(factorial(3))    // → [ 3 ] → [ 2 ] → [ 1 ] → 6
```

Speaker notes

Der Nachteil ist, dass man mit dieser Vorgehensweise keinen Zugriff mehr auf die Originalfunktion hat, da `factorial` selbst verändert wurde (genauer: ergänzt und dem Originalnamen mit Ergänzung zugewiesen). Abhilfe: die Originalfunktion kann als Attribut an die neue Funktion angehängt werden.

```
// verbesserte Version von trace
function trace(func) {
  const f = (...args) => {
    console.log(args)
    return func(...args)
  }
  f.original = func
  return f
}

// 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
// zurück zur ursprünglichen Version
factorial = factorial.original
// Aufruf
console.log(factorial(3)) // → 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.

Speaker notes

Zum Beispiel ist `console.log` sicher keine pure Funktion, da die Ausgabe auf der Konsole ein Seiteneffekt ist. Trotzdem ist `console.log` nützlich.

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 */
```

Speaker notes

Zu beachten: das Array selbst kommt in der Definition von `sum` gar nicht mehr vor. Die Funktion `sum` wird nun durch eine geeignete Kombination zweier Funktionen definiert. Oder anders ausgedrückt: wir stöpseln bestehende Funktionen zu neuen Funktionen zusammen.

Mit dieser `reduce`-Variante können wir auch `append` zur Verkettung von Arrays definieren. Auch `map` und `filter` lassen sich mit `reduce` umsetzen:

```
> const append = (a) => reduce((ls, nx) => [...ls, nx])(a)
> const filter = (f) => reduce((c,n) => f(n) ? [...c,n] : c)([])
> const map = (f) => reduce((c,n) => [...c,f(n)])([])

> append([1,2,3])([10,11,12])
[ 1, 2, 3, 10, 11, 12 ]
> filter(n=>n%2==0)([1,2,3,4,5])
[ 2, 4 ]
> map(n=>2*n)([1,2,3,4,5])
[ 2, 4, 6, 8, 10 ]
```

Currying: Funktion mit mehreren Argumenten wird in Funktionen mit einem Argument zerlegt. Zurückgegeben wird dann jeweils eine Funktion über das nächste Argument. In der Regel geht mit im funktionalen Programmieren davon aus, dass alle Funktionen "curried" sind.

Partielle Anwendung: Eine Funktion mit `n` Argumenten wird mit weniger Argumenten aufgerufen. Zurückgegeben wird eine Funktion über die restlichen Argumente.

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

Speaker notes

Da WBE nur über ein Semester geht, können einige Themen nur kurz angesprochen werden. Dazu gehört die funktionale Programmierung, die vor JavaScript sehr gut unterstützt wird. Zu funktionalem Programmieren in JavaScript gibt es zahlreiche Bücher.

Mehr zu funktionalem Programmieren können Sie in Wahlfächern erfahren:

PSPP (Programmiersprachen und Paradigmen)

G. Burkert und K. Rege

Verschiedene Sprachen (Lisp, Prolog, Smalltalk) und Paradigmen. Einführung in verschiedene Paradigmen, dabei auch mehrere Lektionen und Praktika zum Thema *funktionale Programmierung*.

FUP (Funktionale Programmierung)

D. Flumini

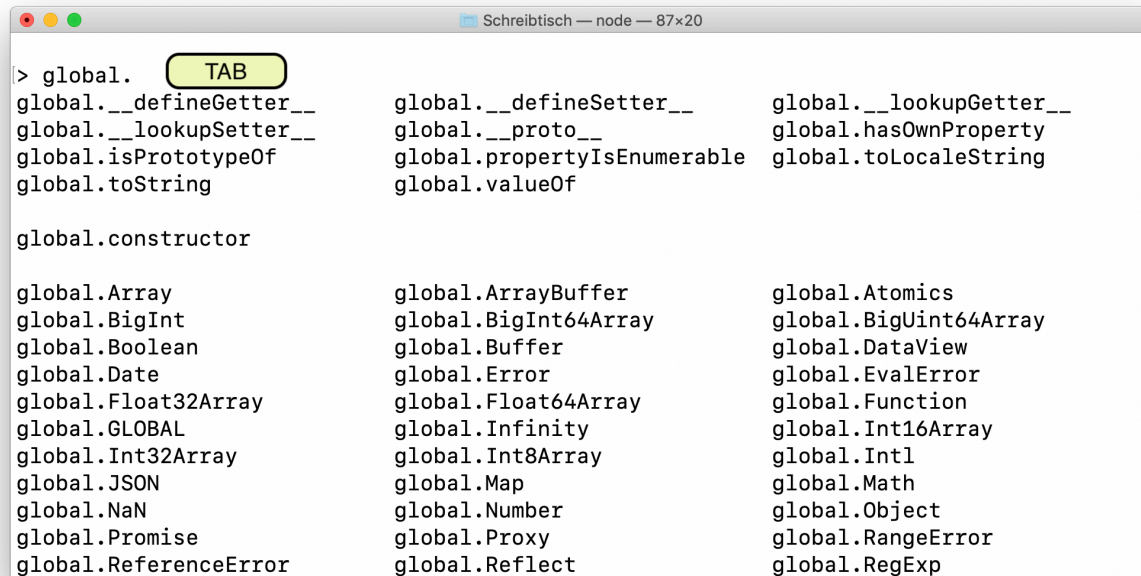
Vertiefte Einführung in die funktionale Programmierung inklusive der theoretischen Grundlagen (Lambda Kalkül). Gut geeignet als Vertiefung von PSPP.

ÜBERSICHT

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

NODE.JS KONSOLE

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



The screenshot shows a terminal window titled "Schreibtisch — node — 87x20". The prompt is "> global.". A yellow box highlights the word "TAB" above the first column of output. The output lists various global objects and methods in three columns. The first column includes methods like __defineGetter__, __lookupSetter__, isPrototypeOf, toString, and constructor, as well as objects like Array, BigInt, Boolean, Date, Float32Array, GLOBAL, Int32Array, JSON, NaN, Promise, and ReferenceError. The second column includes methods like __defineSetter__, __proto__, propertyIsEnumerable, valueOf, ArrayBuffer, BigInt64Array, Buffer, Error, Float64Array, Infinity, Int8Array, Map, Number, Proxy, and Reflect. The third column includes methods like __lookupGetter__, hasOwnProperty, toLocaleString, and objects like Atomics, BigInt64Array, DataView, EvalError, Function, Int16Array, Intl, Math, Object, RangeError, and RegExp.

```
> global.  
global.__defineGetter__  
global.__lookupSetter__  
global.isPrototypeOf  
global.toString  
global.constructor  
  
global.Array  
global.BigInt  
global.Boolean  
global.Date  
global.Float32Array  
global.GLOBAL  
global.Int32Array  
global.JSON  
global.NaN  
global.Promise  
global.ReferenceError  
  
global.__defineSetter__  
global.__proto__  
global.propertyIsEnumerable  
global.valueOf  
global.ArrayBuffer  
global.BigInt64Array  
global.Buffer  
global.Error  
global.Float64Array  
global.Infinity  
global.Int8Array  
global.Map  
global.Number  
global.Proxy  
global.Reflect  
  
global.__lookupGetter__  
global.hasOwnProperty  
global.toLocaleString  
global.Atomics  
global.BigUint64Array  
global.DataView  
global.EvalError  
global.Function  
global.Int16Array  
global.Intl  
global.Math  
global.Object  
global.RangeError  
global.RegExp
```

Speaker notes

Ggf. muss *zweimal* die TAB.Taste gedrückt werden, um die Liste der Attribute und Methoden auszugeben.

Ausserdem gibt es noch die Möglichkeit der Ausgabe mit %o, um ein Objekt zu untersuchen:

```
> console.log( '%o', Number )
```

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
```


Speaker notes

Wenn nur die Argumente benötigt werden, können die ersten beiden Elemente aus dem Array entfernt werden:

```
const args = process.argv.slice(2);
```

Wenn einzelne Argumente geparkt werden sollen, kann *minimist* verwendet werden:

```
const args = require('minimist')(process.argv.slice(2))  
console.log(args['name'])
```

```
$ node app.js name=mustehan dir=/  
mustehan
```

<https://www.npmjs.com/package/minimist>

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>

Speaker notes

Mehr Möglichkeiten bietet das Inquirer.js-Paket, das mit `npm install inquirer` installiert werden kann:

```
const inquirer = require('inquirer');

var questions = [
  {
    type: 'input',
    name: 'name',
    message: "What's your name?"
  }
];

inquirer.prompt(questions).then(answers => {
  console.log('Hi ' + answers['name'] + '!')
});
```

<https://github.com/SBoudrias/Inquirer.js>

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)  
})();
```

Speaker notes

Das hat eigentlich nichts mit Node.js zu tun, es ist eine alte JavaScript-Technik, die bereits verwendet wurde, bevor andere Modulsysteme Einzug gehalten haben. Aber es passt an dieser Stelle ganz gut rein...

MODULSYSTEM (COMMONJS)

```
1 /* car-lib.js */
2 const car = {
3   brand: 'Ford',
4   model: 'Fiesta'
5 }
6
7 module.exports = car
```

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

Speaker notes

- Im Beispiel: `module.exports` exportiert ein Objekt
- Alternative: `exports` exportiert Attribute eines Objekts

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

```
exports.car = car
```

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

Wenn das Argument ein String ist, der nicht wie ein relativer oder absoluter Pfad aussieht, wird angenommen, dass es sich um ein built-in Modul oder ein in `node_modules` installiertes Modul handelt.

```
const readline = require('readline')
```

Noch ein Beispiel:

```
/* reverse.js */
exports.reverse = function(string) {
  return Array.from(string).reverse().join("")
}

/* main.js */
const {reverse} = require("./reverse")

// Index 2 holds the first actual command line argument
let argument = process.argv[2]

console.log(reverse(argument))
```

Aufruf:

```
$ node main.js JavaScript
tpircSavaJ
```

- Das ist das traditionelle Modulsystem in Node.js
- Bezeichnung: *CommonJS*-Modulsystem
- In neueren Versionen wird auch das ES6-Modulsystem unterstützt

Ein Modul ist einfach eine JavaScript-Datei. Ein Modul hat seinen eigenen Scope (Gültigkeitsbereich für Funktionen und Variablen).

Namenskonflikte waren lange ein grosses Problem in JavaScript. Man konnte aber problemlos einfache "Module" herstellen, indem zusammengehörende Funktionen in eine weitere Funktion verpackt werden, welche zurückgibt, was exportiert werden soll. Zum Beispiel hat jQuery seine Funktionalität in einem Objekt `jQuery` verpackt, welches auch über den Namen `$` zugänglich war (sofern `$` nicht bereits für andere Zwecke benötigt wurde).

Für Interessierte:

The Module Pattern, in: Learning JavaScript Design Patterns (Addy Osmani)

<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript>

MODULSYSTEM (ES6)

```
1 /* square.js */
2 const name = 'square'
3 function draw (ctx, length, x, y, color) { ... }
4 function reportArea () { ... }
5
6 export { name, draw, reportArea }
```

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

Speaker notes

- Im Beispiel wurden *named exports* benutzt
- Alternativ gibt es auch *default exports*

```
/* square.js */  
export default function(ctx) {  
  ...  
}
```

```
/* other js file */  
import randomSquare from './modules/square.js';
```

Das mit ES6 in JavaScript eingeführte Modulsystem unterscheidet sich vom in Node.js ursprünglich verwendeten Common.js. Mittlerweile werden in Node.js beide Modulsysteme unterstützt:

<https://nodejs.org/api/esm.html#enabling>

Mehr zu Modulsystemen in JavaScript:

Eloquent JavaScript – Chapter 10 Modules

https://eloquentjavascript.net/10_modules.html

Node Modules at War: Why CommonJS and ES Modules Can't Get Along

<https://redfin.engineering/node-modules-at-war-why-commonjs-and-es-modules-cant-get-along-9617135eeca1>

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
```

Speaker notes

Für die lokale Installation wird das Kommando `npm` von der obersten Ebene des Projektverzeichnisses aus aufgerufen.

Globale Pakete stellen Kommandozeilenbefehle zur Verfügung, die projektübergreifend verwendet werden können.

Beliebte globale Pakete:

- `npm`
- `create-react-app`
- `vue-cli`
- `grunt-cli`
- `react-native-cli`
- `gatsby-cli`
- `nodemon`

Beispiel:

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
  open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
> const {parse} = require("ini");
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

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
```

Speaker notes

Unter *devDependencies* werden Pakete eingetragen, die nur während der Entwicklungsphase benötigt werden.

Aktualisieren von Paketen:

```
npm update
```

```
npm update <package-name>
```

Beispiel für eine `package.json`-Datei

```
{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A Vue.js project",
  "main": "src/main.js",
  "private": true,
  "scripts": {
    "dev": "webpack-dev-server --inline --progress --config build/webpack.dev.conf.js",
    "start": "npm run dev",
    "unit": "jest --config test/unit/jest.conf.js --coverage",
    "test": "npm run unit",
    "lint": "eslint --ext .js,.vue src test/unit",
    "build": "node build/build.js"
  },
  "dependencies": {
    "vue": "^2.5.2"
  },
  "devDependencies": {
    "autoprefixer": "^7.1.2",
    "babel-core": "^6.22.1",
    "...",
    "webpack-merge": "^4.1.0"
  },
  "engines": {
    "node": ">= 6.0.0",
    "npm": ">= 3.0.0"
  },
  "browserslist": [
    "> 1%",
    "last 2 versions",
    "not ie <= 8"
  ]
}
```

Kurzbeschreibung der Einträge

- `name` sets the application/package name
- `version` indicates the current version
- `description` is a brief description of the app/package
- `main` set the entry point for the application
- `private` if set to true prevents the app/package to be accidentally published on npm
- `scripts` defines a set of node scripts you can run
- `dependencies` sets a list of npm packages installed as dependencies
- `devDependencies` sets a list of npm packages installed as development dependencies
- `engines` sets which versions of Node this package/app works on
- `browserslist` is used to tell which browsers (and their versions) you want to support

Quelle: The Node.js Handbook (Flavio Copes)

Seit npm Version 5 wird eine Datei `package-lock.json` angelegt, die die exakten Versionen der installierten Pakete enthält, um ein Projekt wieder mit bestimmten Versionen der verwendeten Pakete herstellen zu können. Das Verzeichnis `node_modules` wird nicht ins Git-Repository übernommen, da es sehr gross sein kann und jederzeit mit den Angaben aus `package.json` und `package-lock.json` wieder angelegt werden kann.

Node.js-Pakete werden semantisch versioniert. Die Versionsnummer besteht aus drei Zahlen:

- Major version
- Minor version
- Patch version

Versionsangaben in package.json:

Versionsangabe	Erklärung	Beispiel
<code>^0.13.0</code>	minor und patch Ebene kann aktualisiert werden	0.13.1, 0.14.0
<code>~0.13.0</code>	patch Ebene kann aktualisiert werden	0.13.1, 0.13.2
<code>>0.13.0</code>	irgendeine neuere Version	0.13.1, 0.14.3
<code>>=0.13.0</code>	diese oder eine neuere Version	0.13.0, 1.0.0
<code>=0.13.3</code>	genau diese Version	0.13.3
<code>2.1.0 – 2.6.2</code>	Versionsbereich	2.2.2, 2.6.2
<code>latest</code>	neueste Version	

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, 3rd Edition
<https://eloquentjavascript.net/>

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

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

