Javascript - Currying

Was ist das?

Beim Currying wird eine Funktion der Ordnung n+1 (also n+1 Parameter) auf eine Funktion der Ordnung n (also n Parameter) reduziert.
Das Konzept dabei ist, die Funktion um einen Parameter zu kürzen und statt des ursprü ngl ichen, berech neten Wertes eine Funktion mit dem gekürzten Parameter zur weiteren Berechnung zurück zu geben. Damit wird die Funktion nun mit einem Parameter weniger aufger ufen. Nun muss aber das Ergebnis noch mit dem reduzi erten Parameter aufgerufen werden. Also in Summe wurden wieder alle Parameter zur Berechnung übergeben, nur verteilt auf 2 Aufrufe oder mehr.

Beispiel für Currying und Uncurrying

```
currye dMu ltiply = (n) => (m) => n * m
currye dMu lti ply (3)(4) === 12 // true
multiply = (n, m) => currye dMu lti ply (n)(m)
multip ly(3, 4) === 12 // true
```

Quelle: https://blog.benestudio.co/currying-in-javascript-es6-540d2a-d09400

Javascript - Ich finde Lambdas verkürzt doof

Javascript - Ich finde Lambdas verkürzt doof (cont)

```
// vollst ändige Schrei bweise mit Selbst aus -
führung am Ende
const berech neW ort laenge = functi on(){
    const map = proces sAr ray (wo erter, function(
wort ){
        return { wort: wort, laenge: wort.l -
engthj};
   });
    con sol e.l og('1. Ausgabe: ' +JSON.s tr ing -
ify (map));
}();
// Umgewa ndelt in lambda Ausdruck
const berech neW ort laenge = functi on(){
    const map = proces sAr ray (wo erter, ( wort )
=>{
          return { wort: wort, laenge: wort.l -
engthj};
   });
    con sol e.l og('2. Ausgabe: ' + JSON.s tri -
ngi fy( map));
}();
// obsolete Klammern entfernt: Parame ter liste,
Body und
// return Schlüs selwort entfernt, da an letzter
Stelle
// => compile Fehler weil das Ergebn iso bjekt als
Body
// interp retiert wird !!!
//const berech neW ort laenge = functi on(){
// const map = proces sAr ray (wo erter, wort =>
// { wort: wort, laenge: wort.l engthj}
// );
// consol e.l og('3. Ausgabe: ' + JSON.s tri ngi -
fy( map));
//}();
// Obsolete Klammern zur Umwandlung in Expression
hinzug efügt
const berech neW ort laenge = functi on(){
    const map = proces sAr ray (wo erter, wort =>
       ({ wort: wort, laenge: wort.l engthj})
   );
    con sol e.l og('4. Ausgabe: ' + JSON.s tri -
ngi fy( map));
}();
```

```
// Lambdas sind so eine schöne Sache aber diese
ständige Verkürzung
// soweit wie geht ist einfach nervig und birgt
Fehler.
// Zwei Klammern für die Parame ter liste und zwei
für den Body kann man
// doch schreiben oder?
// Sie sind anderer Meinung? - is ok.
// Ich nehme an Sie wissen, dass Pfeilf unk tionen
rechts assoziativ sind.
// Trotzdem glaube ich nicht, dass Sie das hier
lesen wollen:
curry = f => a => b => f(a, b)
uncurry = f => (a, b) => f(a)(b)
papply = (f, a) => b => f(a, b)
Quelle: https: //b log.be nes tud io.c o/ cur ryi -
ng- in- jav asc rip t-e s6- 540 d2a d09400
// Anderes und hoffen tlich besseres Beispiel
// Nur zur Vorber eitung
const woerter = ['Hallo', 'heute', 'geste rn'];
const proces sArray = function( items, fn){
    const result = [];
    ite ms.f or Each( (item) => result.pu sh( fn( -
ite m)));
    return result;
}
```

By **Huluvu424242**
(FunThomas424242)

cheatography.com/funthomas424242/
github.com/Huluvu424242

Published 28th April, 2019.
Last updated 2nd July, 2019.

Javascript - Operatoren

a + b	Addition von a und b
a - b	Subtraktion: subtrahiert b von a
a * b	Multiplikation von a und b
x ** y	Exponentialfunktion: x hoch y (wobei sowohl x als auch y variabel sein können - leider keinen korrekten mathematischen Namen gefunden: Potenzfunktion wäre fester Exponent, also auch nicht korrekt.)
a / b	Division: a durch b
a % b	Modulus: a modulo b
a++	Inkrement: erhöht a um 1
a--	Dekrement: verringert a um 1
==	Gleichheitsprüfung nur auf Wertebasis (verschiedene aber wertgleiche Datentypen ergeben true)
===	Gleichheitsprüfung inklusive Werte- und Typgleichheit
!=	Ungleich ohne Typberücksichtigung
!==	Ungleich mit Typberücksichtigung
>	größer als
<	kleiner als
>=	größer gleich
<=	kleiner gleich
?	Ternärer Operator: (bedingung) ? ifAnweisung : elseAnweisung;
&&	logisches UND
||	logisches ODER
!	logisches NOT
typeof <a>	Bestimmt den Typ der Variablen a
o instanceof t	true falls o ein Objekt vom Typ t ist
&	bitweises UND
|	bitweises ODER
~	bitweises NOT
^	bitweises XOR

Javascript - Operatoren (cont)

<<	bitweises links Schieben; mit 0 auffüllend
>>	bitweises rechts Schieben; vorzeichenerhaltend (so die Theorie)
>>>	bitweises rechts Schieben; mit 0 auffüllend

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

Javascript - Datentypen

In Javascript existieren 5 primitive Datentypen:
* String = Zeichenkette
* Number = Ziffernfolge
* Boolean = Boolscher Datentyp
* Null = leere Referenz
* Undefined = Nicht vorhandener Wert

Quelle: https://www.ab-heute-programmieren.de/js-teil-3-datentypen-und-hoisting/

Javascript Konzept: Hoisting

```
Beim Hoisting werden später definierte Variablen bereits am Scopebeginn (für den Entwickler unsichtbar) automatisch deklariert. Dadurch können Wertzuweisungen auf eine Variable erfolgen, welche laut Quellkode noch gar nicht existiert.
*Beispiel
num = 6;
consol e.l og( num); // Gibt 6 zurück
var num = 7;
consol e.l og( num); // Gibt 7 zurück
Hoisting gibt es auch für Funktionen
Beispiel
catNam e("C hlo e");
function catNam e(name) {
    con sol e.l og( "Der Name meiner Katze ist " + name);
}
/*
Das Ergebnis des Codes oben ist: "Der Name meiner Katze ist Chloe"
*/
```

Javascript Konzept: Hoisting (cont)

Wichtig: Es werden nur die Deklar ationen gehoisted, nicht die Zuweis ungen.

Beispiele

```
consol e.l og( num); // Gibt undefined zurück
var num;
num = 6;
var x = 1; // Initia lisiere x
consol e.log(x + " " + y); // '1 undefined'
var y = 2; // Initia lisiere y
// Das obige Beispiel wird implizit als das folgende versta nden:
var x; // Deklariere x
var y; // Deklariere y
// Hoisting beendet.
x = 1; // Initia lisiere x
consol e.log(x + " " + y); // '1 undefined'
y = 2; // Initia lisiere y
```

Quelle: https://developer.mozilla.org/de/docs/Glossary/Hoisting

Javascript - Typische Datentypkonvertierungen

25 + ''	Zahl -> String, Die Zahl 25 wird zu '25'
'25' - 0	String -> Zahl, Der Text '25' wird zur Zahl 25
!!a	any -> boolean, Der Wert der Variablen a wird zum Wahrheitswert: true falls a einen Wert enthält und false falls nicht

Umwandlung einer Promise Callback Hölle

```
Alter Stil:
doSome thi ng( fun cti on( result) {
    doS ome thi ngE lse (re sult, functi on( new -
Result) {
        doT hir dTh ing (ne wRe sult, functi on( -
fin alR esult) {
            con sol e.l og('Got the final result: '
+ finalR esult);
       }, failur eCa llb ack);
   }, failur eCa llb ack);
}, failur eCa llb ack);
Moderner Stil:
doSome thi ng( ).t hen (fu nct ion (re sult) {
    return doSome thi ngE lse (re sult);
```

Umwandlung einer Promise Callback Hölle (cont)

```
})
.then( fun cti on( new Result) {
    return doThir dTh ing (ne wRe sult);
})
.then( fun cti on( fin alR esult) {
    con sol e.l og('Got the final result: ' +
finalR esult);
})
.catch (fa ilu reC all back);
```

Der alte Stil kann in den neuen Stil überführt werden, wenn es nur darum geht, dass die inneren Promises nach den äußeren ausgeführt werden und die äußeren Variablen in den inneren Promises keine Rolle mehr spielen.

https://developer.mozilla.org/de/docs/Web/JavaScript/Guide/Using_promises

Overloading von Funktionssignaturen

```
export class Test{

        public add( a: Number, b:Numb er) :void;
        public add( a: String, b:Stri ng) :void;
        public add( a: boolean, b:bool -
ean ):void;
        public add( a, b):void{
                con sol e.l og( a+b);
       }
        public static run():any{
               let t:Test = new Test();
                t.a dd( 'a' ,'b');
                t.a dd( 3,5);
                t.a dd( tru e,t rue);
       }
}
Test.r un();
```

Erzeugt die Ausgaben:

```
ab
8
2
```