

JAVA SCRIPIT MEMO

Typeof

Mit der Operator `typeof <Wert>` kann man die Datentyp eines Wertes feststellen:

```
1 console.log(typeof "1");
2 console.log(typeof 2);
3 console.log(typeof true);
4 console.log(typeof null); // -> 'object'!
5 console.log(typeof undefined); // -> 'undefined'
```

string
number
boolean
object
undefined

Unäre Operatoren

Die Operatoren `+` und `-` können auch als unäre Operatoren für numerische Werten sinnvoll verwendet werden:

```
1 console.log( - (3-4) );

1
```

Vergleich-Operator

```
1 4 > 3 // → true
2 4 > "3" // → true
3 4 > "5" // → false
4 4 >= "4" // → true
5 4 === 4 // → true
6 4 === '4' // → false
7 0 == {} // → false !
8 0 == [] // → true !
9 4 < {} // → false !
10 4 < [] // → false
11 4 > [] // → true
```

Type Casting

2.5. Type-Casting

Type-Casting bedeutet Abguss. In Informatik bedeutet Type-Casting die Umwandlung von Datentypen von Werten. Einige Einführungsbeispielen:

```
1 -"12" // -12 ('number')
2 -"abc" // NaN ('number')
3 "12" - 1 // 11 ('number')
4 undefined - 1 // NaN ('number')
5 null - 3 // -3 ('number')
6 null - undefined // NaN ('number')
```

Let

Der Befehl `let` deklariert eine neue Variable in dem aktuellen lexikalischen Scope.

```
1 /**
2  * calculate age of father and son, given sum and diff of age of father and son.
3  * @param sum {number} age of father and son
4  * @param diff {number} how old is the father, when he gets the son.
5  */
6 function calculateAge(sum, diff) {
7     let fatherAge = (sum + diff) / 2;
8     let sonAge = fatherAge - diff;
9     return {"father" : fatherAge, "son" : sonAge};
10 }
11 console.log(calculateAge(60, 20)); // → {father: 40, son 20}
12 // console.log(fatherAge) will cause an error
13 {father: 40, son: 20}
```

Hoisting

Deklariert man eine Variable in einer Funktion, hat sie die Scope in der ganze Funktion (Hoisting).

```
1 var data = [2, 3, 5, 7, 11];
2
3 function reverseName( name ) {
4     var data = name.split('').reverse().join(''); // gut practice
5     return data.reverse().join('');
6 }
7
8 reverseOfPeter = reverseName( "Peter" );
9 console.log(reverseOfPeter) // Erhält was man will
10 console.log(data) // Ok, keine Nebeneffekt
11
12 reteP
13 [2, 3, 5, 7, 11]
```

IndexOf

Die Methode `indexOf(<element>)` eines Arrays gibt die Position des zu erst gefundenen Elementes in dem Array welche gleich das gegeben `<element>` ist, zurück, oder `-1`. Zum Vergleichen wird der Operator `===` verwendet.

```
1 let colors = ["blue", "green", "orange", "2", 2];
2 let colorPosition = colors.indexOf("blue"); // 0
3 let numberPosition = colors.indexOf(2); // 4
```

Array als Stack

Stack mit push / pop

```
1 {
2     let colors = ["red", "blue", "orange", "yellow", "white"];
3     // read the last inserted element -> "white"
4     let topColor = colors.pop();
5     console.log(topColor, colors);
6     // insert a new color into the array
7     let numOfElement = colors.push("black");
8     console.log(numOfElement, colors);
9 }
10
11 white ["red", "blue", "orange", "yellow"]
12 5 ["red", "blue", "orange", "yellow", "black"]
```

Array als Queue

Queue mit push/shift

```
// push / shift
{
  let colors = ["red", "blue", "orange"];
  let length = colors.push("black"); // [ 'red', 'blue', 'orange', 'black' ]
  var firstColor = colors.shift(); // [ 'blue', 'orange', 'black' ]
  console.log(firstColor); // red
}
```

Queue mit unshift / pop

```
// unshift / pop
{
  let colors = ["red", "blue", "orange"];
  let length = colors.unshift("white"); // ["white", "red", "blue", "orange"];
  let firstColor = colors.pop(); // [ 'white', 'red', 'blue' ]
  console.log(firstColor); // orange
}
```

Set

Set Operationen

```
1 var colorSet = new Set(['blue', 'red', 'orange', 'red']);
2 colorSet.add('black').add('white');
3 console.log(colorSet.has('blue'),           // true
4               colorSet.has('white'),        // true
5               colorSet.has('yellow') );     // false
6 console.log( colorSet.size );               // 5

true true false
5
```

Sondern Werten

```
1 var falsySet = new Set([0, undefined, null, NaN, 0]);
2 console.log( falsySet.size );
3 console.log( falsySet.has(0),               // true
4               falsySet.has(undefined),      // true
5               falsySet.has(null),           // true
6               falsySet.has(NaN) );          // true
```

Map

Map-Konstruktor

```
1 var textFormat = new Map([
2   ['font-family', 'Helvetica'],
3   ['font-weight', 400],
4   ['color', new Map([
5     ['red', 0x29], ['green', 0x6D], ['blue', 0xFF]
6   ])],
7   ['counter', ['arabic', 'roman', 'alpha', 'greek']]
8 ]);
9 console.log(textFormat.get('font-family'));
10 console.log(textFormat.get('color').get('green'));
```

Neue Element hinzufügen/alte Wert überschreiben:

```
1 textFormat.set('font-size', '12pt');
2 textFormat.set('font-family', 'monospaced');
```

Test ob einen Key existiert

```
1 textFormat.has('font-family');
2 textFormat.has('font-size');
```

Litteral Objekt

Zugriff des Wertes in ein Objekt via Key

```
1 var textFormat = {
2   "font-family" : "Heveltica",
3   "font-weight" : 400,
4   "color": {red: 0x29, green: 0x6D, blue: 0xFF},
5   counter: ["arabic", "roman", "alpha", "greek"]
6 };
7 var fontFamily = textFormat["font-family"];
8 var strong = textFormat["font-weight"];
9 var redComponent = textFormat.color.red;
10 var counterLevel = textFormat.counter[1];
11 var fontSize = textFormat.fontSize ;           // undefined
12 console.log(fontFamily, strong, redComponent, counterLevel);
13 textFormat.fontSize = "12pt";                  // new key/value
14 textFormat.color.red = 0xFF;                   // overwrite
15 console.log(textFormat);

Heveltica 400 41 roman
{font-family: "Heveltica", font-weight: 400, ...}
```

Im Allgemein mit der Methode `hasOwnProperty(<property>)` kann man eine *Property* eines Objekt testen. Es gilt auch für Literal-Objekt:

Test von Existenz einer Property via hasOwnProperty

```
var textFormat = {
  "font-family" : "Helvetica",
  "font-weight" : 400,
  "color": {red: 0x29, green: 0x6D, blue: 0xFF},
  counter: ["arabic", "roman", "alpha", "greek"]
};
console.log(textFormat.hasOwnProperty('font-family')); // true
console.log(textFormat.hasOwnProperty('font-color')); // false
```

For

```
1 let primes = [2, 3, 5, 7, 11];
2 for(let i = 0; i < primes.length; ++i) {
3   console.log( `${i} -> ${primes[i]}` );
4 }
```

```
0 -> 2
1 -> 3
2 -> 5
3 -> 7
4 -> 11
```

Function (Grundlage)

```
// ❶
function buildTriangle(lines,           // ❷
    trailingChar = ' ',               // ❷
    separator= ' ',                   // ❷
    char = '*') {                     // ❷
    var triangle = [];                 // ❸

    for (let i = 0; i < lines; ++i) {
        let line = [];
        for (let j = 1; j < lines - i; ++j) {
            line.push(trailingChar);
        }
        for (let k = 0; k <= i; ++k) {
            line.push(char);
            k < i ? line.push(separator) : null;
        }
        triangle.push(line);
    }
    return triangle;                   // ❸
                                        // ❹
}

// ❶
function printTriangle(triangle) {
    const maxLeading = triangle.length === 0 ? 0 : Math.floor(
Math.log10(triangle.length) );
    for (let index = 0; index < triangle.length; ++index) {
        let leading = index === 0 ? 0 : Math.floor( Math.log10(index) );
        console.log(' '.repeat(maxLeading-leading) + index + '|' +
triangle[index].join(''));
    }
}

var triangle= buildTriangle(11); // ❺
printTriangle(triangle); // ❺
```


Function (OOP)

In diesem Abschnitt haben wir die globale Variable `triangle` zur Verfügung:

```
1 console.log(triangle);
2 /* Die Variable triangle hat diese Struktur:
3 [
4   [" ", " ", " ", " ", " ", " "],
5   [" ", " ", " ", " ", " ", " "],
6   [" ", " ", " ", " ", " ", " "],
7   [" ", " ", " ", " ", " ", " "],
8   [" ", " ", " ", " ", " ", " "],
9 ] */
```

```
printSimpleTriangle
```

Wir betrachte nochmals die Funktion `printSimpleTriangle`:

```
1 function simplePrintTriangle(triangle) {
2     for (let index = 0; index < triangle.length; ++index) {
3         console.log(triangle[index].join(' '));
4     }
5 }
```

Map / forEach : Funktion höherer Order

map

```
function map(triangle, converter){
  let newMappedLine = [];
  for (let i = 0; i < triangle.length; ++i){
    newMappedLine.push (converter(triangle[i]));
  }
  return newMappedLine;
}
```

```
triangle.map( function(line){ return line.join(''); } )  
    .forEach( function(line){ console.log(line)} );
```

forEach

```
function forEach(array, action){
  for (let i = 0; i < triangle.length; ++i){
    action(array[i]);
  }
}
```

Arrow Functions

```
1 var add = (a, b) => a + b;
2 console.log( add(12.3, 45.6) ); // 57.900000000000006
3
4 var quadrad = x => x ** 2;
5 console.log( quadrad(12) ); // 144
6
7 var makeBanner = () => console.log("*$*".repeat(10)); // *$*...*$*
8 makeBanner ();
```

```
57.900000000000006
144
*$$$*$$$*$$$*$$$*$$$*$$$*$$$*$$$*$$$*
```

Array.map

```
1 var stars = LIBRARY.map(element => '*'.repeat(element.star) );
2 console.log(stars);
3 // ["*", "**", "****", "**", "***", "*", "***", ...]
```

Array.forEach

```
1 LIBRARY.forEach(element => console.log( `${element.authors.join()} -> ${element.title}` ) );
David Flanagan -> JavaScript pocket reference
Michel Goossens -> The LaTeX Graphics companion
```

Array.filter

```
1 var bestSeller = LIBRARY.filter(element => element.star === 5 );
2 console.log(bestSeller);
```

```
[{
  key: "ISBN:9780691015149"
  authors: ["Jane Ellen Harrison"]
  title: "Prolegomena to the study of Greek religion"
  publish_date: "1991"
  star: 5
}]
```

Array.reduce

La méthode **reduce()** applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.



JavaScript Demo: Array.reduce()

```
1 const array1 = [1, 2, 3, 4];
2 const reducer = (accumulator, currentValue) => accumulator + currentValue;
3
4 // 1 + 2 + 3 + 4
5 console.log(array1.reduce(reducer));
6 // expected output: 10
7
8 // 5 + 1 + 2 + 3 + 4
9 console.log(array1.reduce(reducer, 5));
10 // expected output: 15
11
```

§ 2.5. Array.reduce

Die Syntax der Funktion `Array.reduce(callback [,initValue])`:

- Die Funktion `callback(accumulator, element, index, array)` hat 4 Argumenten:
 - `accumulator` akkumuliert die Rückgabewerten der Funktion `callback`
 - `element` aktuellen Element der Array.
 - `index` Optional, der Index der Element `element` in der Array
 - `array` Optional, der Array, welche die Funktion `reduce` gehört.
- Das Argument `initValue` kann weggelassen werden wenn der Array nicht leer ist. Das erste Element der Array wird in diesem Fall als `initValue` verwendet.

Die Funktion `recude` ruft die Funktion `callback` für jede Element der Array auf, das Argument `accumulator` ist der Rückgabewert der vorherigen Aufruft der Funktion `callback`.

Summe berechnen

```
1 var integers = [1, 2, 3, 4, 5];
2 var sum = integers.reduce( (a, e) => a + e); // der Init-Wert ist das erste Element von inte
3 console.log(sum); // 15
```

15

Sammlung von alle Autoren in der Array LIBRARY

```
1 var authors = LIBRARY.reduce( (a, e) => a.concat(e.authors), [] ); // Der Initwert muss ange
2 console.log(authors);
```

["David Flanagan", "Michel Goossens", "Alexandre Dumas", ...]

Damit man die duplizierten Autoren aus dem Liste entfernt, verwendet man einen Set:

```
1 var authors = LIBRARY.reduce(
2   (acc, e) => { e.authors.forEach( e => acc.add(e) ); return acc; }
3   , new Set() // (1)
4 );
5 for (a of authors) {
6   console.log(a);
7 }
```

David Flanagan
Michel Goossens
Alexandre Dumas

// : Init-wert = 1 (erste Element von integer)

// : Init-wert muss angegeben werden

Returning Function

3. Returning Function

Eine Funktion in JavaScript kann auch eine Funktion zurückgeben.

```
1 function convertLineFn(triangle){  
2   const maxLeading = triangle.length === 0 ? 0 : Math.floor( Math.log10(triangle.length) )  
3   return function(line, index){  
4     let leading = index === 0 ? 0 : Math.floor( Math.log10(index) ) ;  
5     return ' '.repeat(maxLeading-leading) + index + '|' + line.join('');  
6   }  
7 }  
8 var converter = convertLineFn(triangle);  
9 triangle.map(converter).forEach(l => console.log(l));
```

```
< >  
0|   *  
1|  * *  
2| * * *  
3|* * * *  
4|* * * * *
```

Fehlerbehandlung

Der Befehl throw

```
throw <Object>;
```

Der Befehl try - catch

```
try {  
  <statements> // (1)  
}catch(ex){  
  <statements> // (2)  
}
```

- 1 Die Befehlen hier sind Befehlen, die eine bestimmte Aufgabe ausführen sollen. In einem regulären Ablauf verursacht diesen Befehlen keinen Fehler.
- 2 Wenn der (erste) Fehler in <1> ausgeworfen wird, werden die Befehlen in <2> ausgeführt. Die Variable `ex` referenziert auf dem ausgeworfenen Fehler und steht in der Block 2 zur Verfügung.

```
function buildTriangleWithException(lines,  
    trailingChar = ' ',  
    separator= ' ',  
    char = '*') {  
  let numOfLine = Number(lines);  
  if ( isNaN(numOfLine) ){  
    throw new TypeError(`${lines} is not a number`); // 1  
  }  
  if ( numOfLine < 0 ){  
    throw new RangeError(`Parameter 'line' must be positive, but got ${numOfLine}`);  
  }  
  var triangle = [];  
  
  for (let i = 0; i < numOfLine; ++i) {  
    let line = [];  
    for (let j = 1; j < lines - i; ++j) {  
      line.push(trailingChar);  
    }  
    for (let k = 0; k <= i; ++k) {  
      line.push(char);  
      k < i ? line.push(separator) : null;  
    }  
    triangle.push(line);  
  }  
  return triangle;  
}  
// Nutzung der Funktion:  
try{  
  var triangle = buildTriangleWithException("string"); // 2  
}catch(ex){ // 3  
  console.log(ex);  
}
```

- 1 Ursache des Fehlers.
 - 2 Der Fehler wird ausgeworfen.
 - 3 Der Fehler wird in `catch`-Block abgefangen
-