

Inhalt

1	Algorithmen und Datenstrukturen in ES6+	3
1.1	Warum in ES6+	3
1.2	Nicht behandelte Themen	3
2	Entwicklungsumgebung	5
3	JavaScript Paradigmen	7
3.1	Objekte und Object-Orientiertes Programmieren	7
3.2	Funktionales Programmieren	7
4	JavaScript Konstrukte	9
4.1	Entscheidungskonstrukte	9
4.2	Wiederholungen	9
4.3	Funktionen	9
4.4	Scope	9
4.5	Closures	9
4.6	Call by Reference vs. Call by Value	9
5	Datenstrukturen	11
5.1	Array	11
5.2	Liste	17
5.3	Stack	18
5.4	Queue	18
5.5	Dequeue	18
5.6	Priority Queue	18
5.7	LinkedList	18
5.8	Zirkulare LinkedList	18
5.9	Zweifach verknüpfte LinkedList	18
5.10	Dictionary	18
5.11	Hashing	18
5.12	HashMap	18
5.13	MapTree	18
5.14	LinkedMap	18
5.15	Sets	18

5.16	Binäre Bäume	18
5.17	Graphen	18
5.18	AVL Tree	18
6	Algorithmen	19
6.1	Breitensuche	19
6.2	Tiefensuche	19
6.3	Bubble Sort	19
6.4	Selection Sort	19
6.5	Shellsort	19
6.6	Mergesort	19
6.7	Quicksort	19
6.8	Sequential Suche	19
6.9	Binäres Suchen	19
6.10	Suchen nach Minimum und Maximum	19
6.11	Rucksackproblem	19
6.12	Greedy Algorithm	19

Chapter 1

Algorithmen und Datenstrukturen in ES6+

Dieses Buch ist noch in Bearbeitung....

1.1 Warum in ES6+

- ES is eating the world
 - meaningful understanding
 - I spent most of my professional life writing in JS and I think I know most about it

1.2 Nicht behandelte Themen

- Promises - async/await - webAPI - Browser spezifische Unterschiede

Chapter 2

Entwicklungsumgebung

JS ist eine interpretierte Sprache. Sie läuft auf einer JS Engine. Die JS Engine läuft auf jeden Browser. Man kann direkt im Browser den Code ausführen.

Sobald man den Browser aber neu lädt ist unser Code weg. Besser ist es den Code entweder in nodejs auszuführen.

File speichern und mit `node [filename]` ausführen

Chapter 3

JavaScript Paradigmen

JS ist eine sog. Multiparadigmen Programmiersprache. JS ist imperativ, objektorientiert und funktional.

3.1 Objekte und Object-Orientiertes Programmieren

3.2 Funktionales Programmieren

Chapter 4

JavaScript Konstrukte

4.1 Entscheidungskonstrukte

In JS gibt es if Statements, ternary und switch statement. if if else if else if
ternary operator

Return

statement vs. expression switch

4.2 Wiederholungen

for loop, while loop, symbol iterator; Generator; yields

4.3 Funktionen

4.4 Scope

4.5 Closures

4.6 Call by Reference vs. Call by Value

Chapter 5

Datenstrukturen

5.1 Array

JS hat im Vergleich zu Java oder C/C++ nur sehr wenige Datenstrukturen. Eines ihrer wichtigsten Datenstrukturen ist der Array. Das Array werden wir später dazu benutzen, um alle anderen komplexeren Datenstrukturen zu implementieren. Der Unterschied zu JSs Arrays im Vergleich zu anderen Programmiersprachen ist, dass Arrays in JS keine fixe Länge haben. Durch das Hinzufügen und Entfernen von Elementen verändert sich die Array-Länge dynamisch mit. Bei der Initialisierung muss man dem Array dadurch auch keine bestimmte Länge mitgeben werden.

Arrays können in JS auf zwei Arten erstellt werden: Mit dem Array Konstruktor oder mit Array Literal:

```
1 var myArrayConstructor = new Array();  
2 var myArrayLiteral = [];
```

Listing 5.1: Array Konstruktor

Der Array Konstruktor wird mit `new` eingeleitet und darauf folgt `Array()`. Beim Array Literal wird nur eine eckige Klammer `[]` benötigt. Beide Möglichkeiten erstellen einen Array. Jedoch wird angeraten zur Erstellung eines Arrays das Array Literal zu nehmen. Nicht nur ist er kürzer und auch schneller, er ist auch syntaktisch eindeutiger. Denn mit dem Array Konstruktor kann man auch die Länge des Arrays definieren als auch initialisieren. Die Syntax von beiden Konstrukten sind sich ähnlich, sodass es zu Verwirrung kommen kann, wenn man den Array mit Zahlen initialisiert:

```
1 var myArrayLength = new Array(3);  
2 var myArrayInit = new Array(3,2,1);
```

Listing 5.2: Array Konstruktor

Die Länge eines Arrays wird als Zahl (hier: 3) in den Konstruktor eingeschrieben. Damit hat das Array in unserem Beispiel eine Länge von 3, die man mit `length`

überprüfen kann. Die (hier 3) Elemente sind noch `undefined`, da sie noch nicht initialisiert sind. Bei der Initialisierung gibt man die Elemente ebenfalls in den Konstruktor mit ein, jeweils getrennt durch einen Komma.

```
1 var myArrayLength = new Array(3);
2 console.log(myArrayLength);
3 // [undefined, undefined, undefined]
4 console.log(myArrayLenth.length);
5 // 3
6
7 var myArrayInit = new Array(3,2,1);
8 console.log(myArrayInit);
9 // [3, 2, 1]
10 console.log(myArrayInit.length);
11 // 3
```

Listing 5.3: Array Konstruktor

JS ist nicht static typed. D.h. ein Array kann Elemente nicht nur eines Typen gleichzeitig aufnehmen, sondern auch verschiedene. Ein Array in JS kann damit auch Zahlen, Strings, Bool und Objekte gleichzeitig aufnehmen:

```
1 var myArray = [1, "42", true, "hi", {"hello": "world"}];
```

Listing 5.4: Array Konstruktor

Intern werden die Elemente in einen String gecastet. Dadurch sind Arrays in JS langsamer als in anderen Sprachen. Um auf ein Array-Element zuzugreifen benutzen wir die eckige Klammer `[]`. Ein Array ist Index basiert und fängt mit 0 an. Um auf das zweite Element in einen Array zuzugreifen, schreiben wir also `myArray[1]`:

```
1 var myArray = [1, "42", true, "hi", {"hello": "world"}];
2 console.log(myArray[1]);
3 // "42"
```

Listing 5.5: Array Konstruktor

JS bieten viele Funktionen zur Manipulationen von Arrays an. Zum Hinzufügen am Ende wird `push()` benutzt. Um ein Element am Anfang des Arrays hinzuzufügen, wird `unshift()` verwendet:

```
1 var myArray = [1, 2, 3];
2 myArray.push(4);
3 console.log(myArray);
4 // [1, 2, 3, 4]
5
6 myArray.unshift(0);
7 console.log(myArray);
8 // [0, 1, 2, 3, 4]
```

Listing 5.6: Array Konstruktor

Für das Entfernen am Ende des Arrays gibt es `pop()`. Die Funktion `pop()` entfernt das letzte Element und gibt das entfernte Element zurück. Für das Entfernen am Anfang des Arrays verwendet man `shift()`.

```
1 var myArray = [0, 1, 2, 3, 4];
2 console.log(myArray.pop());
3 // 4
4 console.log(myArray);
5 // [0, 1, 2, 3]
6
7 console.log(myArray.shift());
8 // 0
9 console.log(myArray);
10 // [1, 2, 3]
```

Listing 5.7: Array Konstruktor

Um Elemente hinzuzufügen, zu ersetzen oder zu entfernen, die sich in der Mitte des Arrays befinden, verwendet man `splice()`. `splice()` nimmt als ersten Parameter den Index, an den das neue Element man hinzufügen will. Als zweiten Parameter wieviele Elemente danach ersetzt wird. Alle darauffolgenden Parameter die hinzuzufügenden Elemente.

```
1 var myArray = [1, 3, 4];
2 myArray.splice(1,0,2);
3 // adds a new element, 2, at index 1
4
5 myArray.splice(3,0,5,6,7);
6 // adds new elements, 5,6, and 7, at index 3
7
8 myArray.splice(5, 1);
9 // removes one element at index 5
10
11 myArray.splice(2, 3);
12 // removes 3 elements starting at index 2
13
14 myArray.splice(3, 1, 99);
15 // replaces one element at index 3 with the new element 99
16
17 myArray.splice(3, 2, 42);
18 // replaces 2 elements starting at index 3 with the new element 42
19
20 myArray.splice(999,0,2);
21 // if the index (first parameter) is larger than the array length,
   then it will just adds a new element to the end
```

Listing 5.8: Array Konstruktor

Die Elemente in einen Array kann man mit `sort()` sortieren. `sort()` nimmt eine Methode auf, die zwei Elemente miteinander vergleicht und entweder eine positive oder negative Zahl zurückgibt oder 0. Eine negative Zahl steht dafür, dass die Zahl kleiner ist. Eine positive Zahl steht dafür, dass die Zahl größer ist. Eine 0 steht dafür, dass beide Zahlen gleich sind.

Beim Sortieren von Strings ist keine Funktion notwendig. Jedoch kann man eine mitgeben bei der überprüft wird, ob ein String größer ist als ein anderer String. Wenn man jedoch bei der Sortierung von Zahlen sich auf die Default `sort()` Funktion verlässt, dann können die Zahlen falsch sortiert werden:

```

1 var myArray = [1, 5, 1001, 8, 4];
2 myArray.sort((a,b) => a - b);
3 console.log(myArray);
4 // [1, 4, 5, 8, 1001]
5
6 var myArray = [1, 5, 1001, 8, 4];
7 myArray.sort();
8 console.log(myArray);
9 // [1, 1001, 4, 5, 8]
10
11 var myArray = ["a", "c", "xxx", "bd"];
12 myArray.sort();
13 console.log(myArray);
14 // ["a", "bd", "c", "xxx"]
15
16 var myArray = ["a", "c", "xxx", "bd"];
17 myArray.sort((a, b) => a > b ? 1 : -1);
18 console.log(myArray);
19 // ["a", "bd", "c", "xxx"]

```

Listing 5.9: Array Konstruktor

Man kann die `sort()` Methode auch benutzen, um ein Array absteigend zu sortieren. Dazu kehrt man das Vorzeichen der mitzugebenden Funktion um:

```

1 var myArray = [1, 5, 1001, 8, 4];
2 myArray.sort((a,b) => b - a);
3 console.log(myArray);
4 // [1001, 8, 5, 4, 1]
5
6 var myArray = ["a", "c", "xxx", "bd"];
7 myArray.sort((a, b) => (a > b ? -1 : 1));
8 console.log(myArray);
9 // ["xxx", "c", "bd", "a"]

```

Listing 5.10: Array Konstruktor

Alternativ kann man zur Umkehrung des Arrays auch die `reverse()` Funktion benutzen. Diese ist sogar etwas schneller als nur die Sort Funktion mit der Methode und dem umgekehrten Vorzeichen von oben zu benutzen.

```

1 var myArray = [1, 5, 1001, 8, 4];
2 myArray.sort((a,b) => a - b).reverse();
3 console.log(myArray);
4 // [1001, 8, 5, 4, 1]
5
6 var myArray = ["a", "c", "xxx", "bd"];
7 myArray.sort().reverse();
8 console.log(myArray);

```

```
9 // ["xxx", "c", "bd", "a"]
```

Listing 5.11: Array Konstruktor

Die Funktionen `push()`, `unshift()`, `pop()`, `shift()`, `splice()`, usw. verändern den originären Array. Es gibt auch Funktionen, die das Original nicht ändert, sondern stattdessen einen neuen Array zurückliefert.

Die `map()` Methode geht durch die Elemente durch und wendet dabei eine ihr mitgegebene Methode (`currentValue, index, array`)=> {} auf jedes einzelne Element im Array an. Die ihr mitgegebene Methode nimmt als erstes Argument das aktuelle Element auf. Das zweite Element ist das momentane Index im Array. Das dritte Argument das ursprüngliche Array. Das Ergebnis ist wieder ein Array mit derselben Länge, wie das ursprüngliche Array:

```
1 var myArray = [1, 2, 3, 4];
2 function multiplyBy2 = item => item * 2;
3 var doubleArray = myArray.map(multiplyBy2);
4 console.log(doubleArray);
5 // [2, 4, 6, 8]
```

Listing 5.12: Array Konstruktor

Die `filter()` Methode nimmt ebenfalls eine Funktion als Argument auf und wendet sie auf alle Elemente im Array an. Nur wenn die Auswertung dieser Funktion auf das aktuelle Element `truthy` zurückgibt, wird dieses Element auch Teil des später zurückgegebenen Arrays. Die ihr übergebene Methode (`currentValue, index, array`)=> {} hat als erstes Argument das aktuelle Element. Das zweite Argument ist der aktuelle Index. Das dritte Argument das original Array auf das die `filter()` Methode angewendet wird. Nur das erste Argument ist verpflichtend. Die restlichen sind optional. Beispielhaft ist unten die `filter()` Methode zum Filtern von nur geraden Zahlen gezeigt:

```
1 var myArray = [1, 2, 3, 4, 5, 6];
2 function isEven = item => item % 2 ;
3 var onlyEven = myArray.map(isEven);
4 console.log(onlyEven);
5 // [2, 4, 6]
```

Listing 5.13: Array Konstruktor

Die `reduce()` Methode unterscheidet sich von `map()` und `filter()` dadurch, dass nicht immer ein Array zurück gegeben werden muss. Stattdessen reduziert die Methode das Array auf einen einzigen Wert. Dieser Wert kann eine Zahl oder auch ein Array sein. Als erstes Argument kann sie eine Methode nehmen. Als zweites Argument nimmt sie einen initial Wert an. Das zweite Argument ist nur optional. Die Methode, die sie aufnimmt, (`acc, item, index, array`)=> {} hat als erstes Argument einen Akkumulator, der den kumulierten Wert enthält und der am Ende das Ergebnis darstellt. Das zweite Argument ist der aktuelle Wert. Das dritte Argument der Index und das vierte das original Array. Nur die ersten beiden Argumente sind verpflichtend.

```

1 var myArray = [1, 2, 3];
2 var sum = (acc, item) => acc + item;
3 var sumOfArray = myArray.reduce(sum, 0);
4 // 6

```

Listing 5.14: Array Konstruktor

Die Methode `flat()`¹ wird auf Arrays angewendet, die selbst wiederum Arrays enthalten. Sie erstellt ein neues Arrays mit allen Unterarrays. Dabei kann optional bestimmt werden bis zu welcher Ebene die Unterarrays aufgelöst werden. Die `flat()` Methode nimmt optional nur einen numerischen Wert an. Dieser legt fest bis zu welcher Ebene die Unterarrays aufgelöst werden.

```

1 var myArray = [1, 2, 3, [4, 5, 6]];
2 console.log(myArray.flat());
3 // [1, 2, 3, 4, 5, 6]
4
5 var myArray = [1, 2, 3, [4, [5, 6]]];
6 console.log(myArray.flat());
7 // [1, 2, 3, 4, [5, 6]]
8
9 var myArray = [1, 2, 3, [4, [5, 6]]];
10 console.log(myArray.flat(2));
11 // [1, 2, 3, 4, 5, 6]

```

Listing 5.15: Array Konstruktor

Die `flat()` Methode wird auch genutzt, um leere Elemente in Arrays zu entfernen

```

1 var myArray = [1, 2, 3, , ,6];
2 console.log(myArray.flat());
3 // [1, 2, 3, 6]

```

Listing 5.16: Array Konstruktor

Die Methode `flatMap()`² ist identisch zum Aufruf einer `map()` Methode gefolgt vom Aufruf einer `flat()` Methode. Die Methode wendet eine ihr mitgegebene Funktion auf alle Elemente an und flacht sie anschließend ab. Die ihr mitgegebene Funktion `(item, index, array)=> {}` nimmt als erstes Argument das aktuelle Element. Das zweite Argument der Index und das dritte Argument das Original Array.

```

1 var myArray = [1, 2, 3];
2 var duplicate = item => [item, item];
3 console.log(myArray.flatMap(duplicate));
4 // [1, 1, 2, 2, 3, 3]

```

Listing 5.17: Array Konstruktor

Die Funktion `concat()` vereint zwei Arrays und liefert das vereinigte Array als neues Array wieder zurück:

¹noch im Experiment Status, d.h. es wurde noch nicht von allen JS Engines implementiert

²noch im Experimentier Status, d.h. nicht alle Browser haben es implementiert


```

1 var myArray1 = ["a", "b", "c", "x"];
2 var myArray2 = ["d", "e", "f"];
3 var newArray = myArray1.concat(myArray2);
4 console.log(newArray);
5 // ["a", "b", "c", "x", "d", "e", "f"]

```

Listing 5.18: Array Konstruktor

Die `slice()` Methode

`every()` `some()`

Array kopieren.

Element im Array finden

Strings in einen Array umwandeln

Array manipulieren: Hinzufügen (`push`, `unshift`), entfernen (des letzten `pop`, des ersten `shift`), Zugriff die Elemente. Subarray erstellen: `splice` (mit mutation), `slice` (ohne). Kopieren von Arrays (`slice(0)`), usw..... dieser Abschnitt eng in Abstimmung mit der Implementierung der komplexen Datenstrukturen machen.

- arrays are only objects- values are converted to Strings- is slower than in other JS- creating an Array with Array literal or constructor - go with the array literal because it's faster and is more clear. Array constructor can be weird....e.g.: `new Array(10);` // creates new array of size 10 `new Array(10, 9, 8, 7, 6);` // creates new array with the elements 10, 9, 8, etc.- different objects, e.g. strings, int, bool can be in one array- accessing array elements- writing into array- can grow dynamically beyond the specified size- creating arrays from strings with `split()`- Assign array to another, shallow copy- How to copy array- searching for an value, `indexOf()`, return the first element, `lastIndexOf()`, returns the last element- exists value, `includes()`- return string from an array: `toString()`, `join()`- creating new arrays from existing ones: `concat()`, `splice()`- mutator functions: `shift()`, `push()`, `pop()`, - reordering `reverse()`, `sort()`; (`sort` doesn't work well with numbers!)- adding and removeing elements from the middle of an array: `splice()`- iterator functions: `forEach`, `every()`, `some()`, `map()`, `filter()`, `reduce()`, `reduceRight()`, `flat()`, `flatMap()`- multidimensional Array

5.2 Liste

asldfjlkewjf Lists are convenient if the order doesn't matter or if you don't have to search for a certain item implement a list
`listSize` (property) Number of elements in list
`pos` (property) Current position in list
`length` (property) Returns the number of elements in list
`clear` (function) Clears all elements from list
`toString` (function) Returns string representation of list
`getElement` (function) Returns element at current position
`insert` (function) Inserts new element after existing element
`append` (function) Adds new element to end of list
`remove` (function) Removes element from list
`front` (function) Sets current position to first element of list
`end` (function) Sets current position to last element of list
`prev` (function) Moves current position back one element
`next` (function) Moves current position forward one element
`currPos` (function) Returns the current

position in list moveTo (function) Moves the current position to specified position

While writing also test your implementation with console.assert()

5.3 Stack

5.4 Queue

5.5 Dequeue

5.6 Priority Queue

5.7 LinkedList

5.8 Zirkulare LinkedList

5.9 Zweifach verknüpfte LinkedList

5.10 Dictionary

5.11 Hashing

5.12 HashMap

5.13 MapTree

5.14 LinkedMap

5.15 Sets

5.16 Binäre Bäume

5.17 Graphen

5.18 AVL Tree

Chapter 6

Algorithmen

Sortier Algorithmen und Such Algorithmen

6.1 Breitensuche

6.2 Tiefensuche

6.3 Bubble Sort

6.4 Selection Sort

6.5 Shellsort

6.6 Mergesort

6.7 Quicksort

6.8 Sequential Suche

6.9 Binäres Suchen

6.10 Suchen nach Minimum und Maximum

6.11 Rucksackproblem

6.12 Greedy Algorithm

```
1 Name.prototype = {  
2   methodName: function(params){  
3     var doubleQuoteString = "some text";  
4     var singleQuoteString = 'some more text';  
5     // this is a comment  
6     if(this.confirmed != null && typeof(this.confirmed) == Boolean  
7       && this.confirmed == true){  
8       document.createElement('h3');  
9       $('#system').append("This looks great");  
10      return false;  
11    } else {  
12      throw new Error;  
13    }  
14  }
```

Listing 6.1: My Javascript Example

laskd flwkejf wlekjf ew