3.4 – JSA: Arrays und ihre Methoden

Einleitung

Arrays sind in der modernen JavaScript-Entwicklung unverzichtbar. Sie dienen dazu, mehrere Werte in einer geordneten Liste zu speichern und bieten eine Vielzahl an integrierten Methoden zur Bearbeitung dieser Daten. In diesem Kapitel werden alle relevanten Array-Methoden systematisch und tiefgehend behandelt, orientiert an den Inhalten der JSA-Zertifizierung und praxisnah aufbereitet.

---

1. Erstellung von Arrays

1.1 Mit Literalen

```
const a = [];
const b = [1, 2, "drei"];
```

1.2 Mit dem Konstruktor `new Array()`

```
const c = new Array();           // []
const d = new Array(4);          // [undefined, undefined, undefined, undefined]
const e = new Array(1, 2, 3);    // [1, 2, 3]
```

Hinweis: Wenn nur ein numerischer Parameter übergeben wird, interpretiert JavaScript diesen als Länge des Arrays. Sonst wird ein Array mit den übergebenen Werten erstellt.

---

2. Zugriff und Indexierung

2.1 Zugriff über den Index

```
const arr = [10, 20, 30];
console.log(arr[0]); // 10
console.log(arr[5]); // undefined
```

2.2 Die Eigenschaft `length`

Beschreibung: Gibt die Anzahl der Elemente im Array zurück.

```
console.log(arr.length); // 3
```

2.3 Methode `at()` (ES2022)

Syntax:

```
array.at(index)
```

Parameter:

- `index` (Zahl): Der Index. Negative Werte zählen vom Ende des Arrays.

Beispiele:

```
arr.at(0);   // 10
arr.at(-1);  // 30 (letztes Element)
```

---

3. Elemente hinzufügen und entfernen (überarbeitet)

3.1 `push()`

Beschreibung: Fügt ein oder mehrere Elemente am Ende eines Arrays hinzu. Das Ursprungsarray wird verändert.

Syntax:

```
array.push(element1, ..., elementN)
```

Parameter:

Parameter	Typ	Beschreibung
element1… N	beliebig	Ein oder mehrere Elemente, die am Ende angehängt werden

Rückgabewert:

- Neue Länge des Arrays nach dem Einfügen.

Seiteneffekt:

- Verändert das Originalarray **in-place**.

Beispiele:

```
const arr = [1, 2];
arr.push(3);               // → [1, 2, 3], Rückgabewert: 3
arr.push(4, 5);            // → [1, 2, 3, 4, 5], Rückgabewert: 5
```

3.2 `unshift()`

Beschreibung: Fügt ein oder mehrere Elemente am Anfang eines Arrays hinzu. Das Ursprungsarray wird verändert.

Syntax:

```
array.unshift(element1, ..., elementN)
```

Parameter:

Parameter	Typ	Beschreibung
element1… N	beliebig	Elemente, die am Anfang eingefügt werden

Rückgabewert:

- Neue Länge des Arrays.

Seiteneffekt:

- Verändert das Originalarray **in-place**.

Beispiele:

```
let arr = [10, 20];
arr.unshift(5);            // → [5, 10, 20], Rückgabewert: 3
arr.unshift(0, 1);         // → [0, 1, 5, 10, 20], Rückgabewert: 5
```

3.3 `pop()`

Beschreibung: Entfernt das letzte Element eines Arrays.

Syntax:

```
array.pop()
```

Parameter: Keine

Rückgabewert:

- Das entfernte Element (oder `undefined`, wenn das Array leer ist).

Seiteneffekt:

- Verändert das Originalarray **in-place**.

Beispiel:

```
const arr = [1, 2, 3];
const last = arr.pop();   // → last: 3, arr: [1, 2]
```

3.4 `shift()`

Beschreibung: Entfernt das erste Element eines Arrays.

Syntax:

```
array.shift()
```

Parameter: Keine

Rückgabewert:

- Das entfernte Element (oder `undefined`, wenn das Array leer ist).

Seiteneffekt:

- Verändert das Originalarray **in-place**.

Beispiel:

```
const arr = [10, 20, 30];
const first = arr.shift();   // → first: 10, arr: [20, 30]
```

3.5 Direkte Indizierung

Beschreibung: Über direkte Indexzuweisung lassen sich Werte gezielt an eine bestimmte Position setzen. Dabei kann das Array "gestreckt" werden.

Beispiel:

```
const arr = [1, 2];
arr[4] = 99; // → [1, 2, <2 leere Slots>, 99]
```

Achtung: Leere Slots ≠ `undefined`. Sie sind „nicht initialisiert“.

3.6 `delete`-Operator

Beschreibung: Entfernt ein Element aus einem Array, ohne die Länge zu ändern.

Syntax:

```
delete array[index]
```

Rückgabewert:

- `true`, wenn erfolgreich (auch wenn Slot leer bleibt)

Seiteneffekt:

- Erzeugt ein „leeres“ Element (`empty`) an der Position

Beispiel:

```
const arr = [1, 2, 3];
delete arr[1]; // → [1, <1 empty slot>, 3]
```

Best Practice: Vermeide `delete` bei Arrays. Nutze `splice()` oder `filter()`.

---

4. Teile extrahieren und kopieren (überarbeitet)

4.1 `slice()`

Beschreibung: Gibt einen Ausschnitt eines Arrays als neues Array zurück. Das Originalarray bleibt unverändert.

Syntax:

```
array.slice(beginIndex[, endIndex])
```

Parameter:

Parameter	Typ	Beschreibung
beginIndex	Zahl	Startindex (inklusive). Negativ → vom Ende gezählt
endIndex	Zahl	Endindex (exklusiv). Optional. Negativ → vom Ende gezählt

Rückgabewert:

- Ein neues Array mit den kopierten Elementen.

Seiteneffekt:

- Kein Seiteneffekt. Das Originalarray bleibt erhalten.

Beispiele:

```
const arr = ["a", "b", "c", "d", "e"];
arr.slice(1, 4);    // ["b", "c", "d"]
arr.slice(2);       // ["c", "d", "e"]
arr.slice(-2);      // ["d", "e"]
arr.slice(1, -1);   // ["b", "c", "d"]
```

4.2 `splice()`

Beschreibung: Fügt Elemente ein, entfernt sie oder ersetzt sie an beliebiger Position im Array. Verändert das Originalarray.

Syntax:

```
array.splice(start, deleteCount[, item1[, item2[, ...]]])
```

Parameter:

Parameter	Typ	Beschreibung
start	Zahl	Startindex der Veränderung
deleteCount	Zahl	Anzahl zu entfernender Elemente ab `start`
item1… N	beliebig	Elemente, die ab `start` eingefügt werden (optional)

Rückgabewert:

- Ein neues Array mit den entfernten Elementen.

Seiteneffekt:

- Das Originalarray wird **in-place** verändert.

Beispiele:

```
const arr = [10, 20, 30, 40, 50];
arr.splice(2, 1);              // → [30], arr = [10, 20, 40, 50]
arr.splice(1, 2, 99, 100);     // → [20, 40], arr = [10, 99, 100, 50]
arr.splice(2, 0, 77);          // → [], Einfügen ohne Entfernen
```

Hinweis:

- Nutze `splice(0)` oder `splice(0, arr.length)` um ein Array zu leeren.

4.3 `fill()`

Beschreibung: Überschreibt (einen Teil von) Array-Elementen mit einem statischen Wert. Verändert das Originalarray.

Syntax:

```
array.fill(value[, start[, end]])
```

Parameter:

Parameter	Typ	Beschreibung
value	beliebig	Der zu schreibende Wert
start	Zahl	Startindex (inklusive, optional, Standard: 0)

Parameter	Typ	Beschreibung
end	Zahl	Endindex (exklusiv, optional, Standard: array.length)

Rückgabewert:

- Das veränderte Originalarray (Referenz).

Seiteneffekt:

- Das Originalarray wird überschrieben (in-place).

Beispiele:

```
const arr = [1, 2, 3, 4, 5];
arr.fill(0);               // → [0, 0, 0, 0, 0]
arr.fill(7, 1, 4);         // → [0, 7, 7, 7, 0]
new Array(3).fill("?");   // → ["?", "?", "?"]
```

---

5. Arrays kombinieren und zusammenfügen (überarbeitet)

5.1 `concat()`

Beschreibung: Kombiniert ein oder mehrere Arrays zu einem neuen Array. Das Originalarray bleibt unverändert.

Syntax:

```
array.concat(array2[, array3, ...])
```

Parameter:

Parameter	Typ	Beschreibung
array2… N	Array	Arrays oder Werte, die angehängt werden

Rückgabewert:

- Ein neues Array mit den kombinierten Werten.

Seiteneffekt:

- Kein. Das Originalarray wird **nicht** verändert.

Beispiele:

```
const a = [1, 2];
const b = [3, 4];
const c = a.concat(b);     // → [1, 2, 3, 4]
const d = a.concat(5, 6);  // → [1, 2, 5, 6]
```

5.2 Spread-Operator `...`

Beschreibung: Entpackt die Elemente eines Arrays an einer bestimmten Stelle (z. B. in Literalen oder Funktionsaufrufen).

Beispiele:

```
const a = [1, 2];
const b = [3, 4];
```

```
const merged = [...a, 99, ...b]; // → [1, 2, 99, 3, 4]

const sum = (x, y, z) => x + y + z;
const values = [10, 20, 30];
sum(...values); // → 60
```

Hinweis: Der Spread-Operator ersetzt keine Methode, sondern ist Teil der Syntax.

5.3 `Array.from()`

Beschreibung: Erstellt ein echtes Array aus einem "array-ähnlichen" oder iterable Objekt (z. B. Strings, NodeLists).

Syntax:

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

Parameter:

Parameter	Typ	Beschreibung
arrayLike	iterable	Objekt mit .length oder Iterator
mapFn	Funktion	Optional: Mapping-Funktion f. jedes Element
thisArg	beliebig	Optional: Wert für `this` innerhalb von mapFn

Beispiele:

```
Array.from("test");            // → ["t", "e", "s", "t"]
Array.from([1, 2, 3], x => x * 2); // → [2, 4, 6]
```

---

6. Iteration: Schleifen und `forEach()` (überarbeitet)

6.1 `forEach()`

Beschreibung: Führt eine Callback-Funktion für jedes Element des Arrays aus.

Syntax:

```
array.forEach(callback(element[, index[, array]])[, thisArg])
```

Parameter:

Parameter	Typ	Beschreibung
callback	Funktion	Wird für jedes Element aufgerufen
thisArg	beliebig	Optionaler Kontext für `this` innerhalb der Funktion

Rückgabewert:

- `undefined`

Seiteneffekt:

- Wird zur Nebenwirkungsausführung genutzt. Verändert nicht das Array.

Beispiele:

```
const arr = [10, 20, 30];
arr.forEach((el, idx) => {
  console.log(`Index ${idx}: ${el}`);
});
```

Vergleich mit Schleifen:

```
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

Hinweis: `forEach()` ist nicht unterbrechbar (`break` / `return` beenden nur die Callback-Funktion).

---

7. Suche und Bedingungsprüfung (überarbeitet)

7.1 `includes()`

Beschreibung: Prüft, ob ein Array einen bestimmten Wert enthält.

Syntax:

```
array.includes(valueToFind[, fromIndex])
```

Parameter:

Parameter	Typ	Beschreibung
valueToFind	beliebig	Gesuchter Wert
fromIndex	Zahl	Index, ab dem gesucht wird (optional)

Rückgabewert:

- `true` wenn Wert enthalten, sonst `false`

Beispiel:

```
[1, 2, 3].includes(2);       // true
[1, 2, 3].includes(4);       // false
[1, 2, 3].includes(1, 1);    // false
```

7.2 `indexOf()` / `lastIndexOf()`

Beschreibung: Liefert den Index des ersten bzw. letzten Vorkommens eines Werts im Array.

Syntax:

```
array.indexOf(searchElement[, fromIndex])
array.lastIndexOf(searchElement[, fromIndex])
```

Beispiel:

```
const items = ["a", "b", "a"];
items.indexOf("a");      // 0
```

```
items.lastIndexOf("a");  // 2
```

Hinweis: Gibt `-1` zurück, wenn nicht gefunden.

7.3 `find()` / `findIndex()`

Beschreibung: Durchsucht ein Array anhand einer Bedingung (Callback).

Syntax:

```
array.find(callback(element[, index[, array]])[, thisArg])
array.findIndex(callback(element[, index[, array]])[, thisArg])
```

Parameter:

- `callback`: Funktion, die für jedes Element `true` oder `false` zurückgibt.
- `thisArg`: Optionaler Kontext für die Callback-Funktion.

Beispiele:

```
const arr = [5, 12, 8, 130, 44];
arr.find(el => el > 10);       // 12
arr.findIndex(el => el > 100); // 3
```

7.4 `some()` / `every()`

Beschreibung: Prüft, ob **mind. ein** (`some`) bzw. **alle** (`every`) Elemente eine Bedingung erfüllen.

Syntax:

```
array.some(callback(element[, index[, array]])[, thisArg])
array.every(callback(element[, index[, array]])[, thisArg])
```

Beispiele:

```
const numbers = [10, 20, 30];
numbers.some(x => x > 15); // true
numbers.every(x => x > 15); // false
```

---

8. Transformation von Arrays (überarbeitet)

8.1 `map()`

Beschreibung: Erstellt ein neues Array, indem auf jedes Element eine Funktion angewendet wird.

Syntax:

```
array.map(callback(element[, index[, array]])[, thisArg])
```

Parameter:

- `callback`: Funktion zur Erzeugung der neuen Werte

- `thisArg`: Optionaler Kontext für die Funktion

Beispiele:

```
const numbers = [1, 2, 3];
const squared = numbers.map(n => n * n); // [1, 4, 9]
```

8.2 `filter()`

Beschreibung: Gibt ein neues Array mit allen Elementen zurück, für die `callback` `true` ergibt.

Syntax:

```
array.filter(callback(element[, index[, array]])[, thisArg])
```

Beispiel:

```
const data = [5, 12, 18, 7];
const filtered = data.filter(n => n > 10); // [12, 18]
```

8.3 `flatMap()`

Beschreibung: Wie `map()`, aber das Ergebnis wird flach gemacht (eine Ebene entfernt).

Syntax:

```
array.flatMap(callback(element[, index[, array]])[, thisArg])
```

Beispiel:

```
const arr = [1, 2, 3];
arr.flatMap(n => [n, n * 2]); // [1, 2, 2, 4, 3, 6]
```

---

9. Aggregation mit `reduce()` (überarbeitet)

9.1 `reduce()`

Beschreibung: Reduziert ein Array durch sukzessives Anwenden einer Funktion auf einen einzigen Rückgabewert. Besonders geeignet für Summen, Objekte, komplexe Aggregationen.

Syntax:

```
array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])
```

Parameter:

Parameter	Typ	Beschreibung
callback	Funktion	Reduktionsfunktion, die den Akkumulator aktualisiert
accumulator	beliebig	Zwischenergebnis der Reduktion

Parameter	Typ	Beschreibung
currentValue	beliebig	Aktuelles Element
index (optional)	Zahl	Index des aktuellen Elements
array (optional)	Array	Ursprungsarray
initialValue	beliebig	Anfangswert des Akkumulators (optional, aber empfohlen)

Rückgabewert:

- Der Endwert der Reduktion.

Beispiele:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, val) => acc + val, 0); // → 10

// Objekt mit Indizes als Werte erzeugen:
const keys = ["a", "b", "c"];
const result = keys.reduce((obj, key, index) => {
  obj[key] = index;
  return obj;
}, {}); // → { a: 0, b: 1, c: 2 }
```

Hinweis: Ohne `initialValue` wird der erste Array-Wert als Startwert verwendet, die Reduktion beginnt bei Index 1.

---

10. Sortierung und Umkehrung (überarbeitet)

10.1 `sort()`

Beschreibung: Sortiert die Elemente eines Arrays **in-place** anhand einer optionalen Vergleichsfunktion.

Syntax:

```
array.sort([compareFunction])
```

Parameter:

Parameter	Typ	Beschreibung
compareFunction	Funktion	Optional. Entscheidet über Sortierlogik

Vergleichsfunktion:

```
(a, b) => a - b   // aufsteigend
(a, b) => b - a   // absteigend
```

Rückgabewert:

- Referenz auf das sortierte Originalarray

Beispiele:

```
const nums = [10, 2, 30];
nums.sort();              // → [10, 2, 30] (alphabetisch!)
nums.sort((a, b) => a - b); // → [2, 10, 30]
```

```
const words = ["Zebra", "Apfel", "äpfel"];
words.sort((a, b) => a.localeCompare(b));
```

Hinweis:

- Ohne `compareFunction` sortiert JavaScript nach UTF-16-Zeichenfolge!

10.2 `reverse()`

Beschreibung: Kehrt die Reihenfolge der Array-Elemente um. Verändert das Array **in-place**.

Syntax:

```
array.reverse()
```

Parameter: Keine

Rückgabewert:

- Referenz auf das umgekehrte Array (Originalarray)

Beispiele:

```
const arr = [1, 2, 3];
arr.reverse();    // → [3, 2, 1]
arr.reverse();    // → [1, 2, 3] (wieder zurück)
```

---

11. Weitere Array-Methoden und Eigenschaften (überarbeitet)

11.1 `Array.isArray()`

Beschreibung: Prüft, ob ein Wert tatsächlich ein Array ist (z. B. nützlich bei API-Rückgaben oder Typsicherheit).

Syntax:

```
Array.isArray(value)
```

Rückgabewert:

- `true` wenn `value` ein echtes Array ist, sonst `false`

Beispiele:

```
Array.isArray([1, 2, 3]);   // true
Array.isArray("text");      // false
Array.isArray({ 0: "a", length: 1 }); // false
```

11.2 `copyWithin()`

Beschreibung: Kopiert einen Teil eines Arrays an eine andere Position **im selben Array**, ohne dessen Länge zu verändern. Die Zielposition wird überschrieben.

Syntax:

```
array.copyWithin(target, start[, end])
```

Parameter:

Parameter	Typ	Beschreibung
target	Zahl	Zielindex, an den kopiert wird (Überschreibung beginnt dort)
start	Zahl	Startindex des zu kopierenden Abschnitts
end	Zahl	(optional) Ende (exklusiv) des Abschnitts

Rückgabewert:

- Referenz auf das veränderte Array

Seiteneffekt:

- Verändert das Ursprungsarray **in-place**.

Beispiele:

```
let arr = [1, 2, 3, 4, 5];
arr.copyWithin(0, 3);        // → [4, 5, 3, 4, 5]
// Erklärung: [3, 4, 5] wird auf Index 0 geschrieben → überschreibt [1, 2, 3]

arr = [1, 2, 3, 4, 5];
arr.copyWithin(1, 0, 2);     // → [1, 1, 2, 4, 5]
// Erklärung: Bereich [0,2) → [1,2] → ab Index 1 eingefügt
```

Tipp:

- Gut für Low-Level-Manipulation, z. B. in Performance-Szenarien oder bei Buffers
- Nicht für semantisches Einfügen (→ nutze `splice()` oder `slice()`)

---

12. Best Practices

- Nutze `filter()`, `map()`, `reduce()` für **pure Funktionen ohne Seiteneffekt**
- Verwende `spread` und `slice()` für **nicht-destruktive Kopien**
- Vermeide `delete` bei Arrays – es erzeugt "Löcher"
- Setze `sort()` immer mit Vergleichsfunktion ein (numerisch/alphabetisch)
- Bei tief verschachtelten Arrays: `flat(depth)` bewusst einsetzen

---

13. Übungsaufgaben (ergänzt mit Lernzielen)

1. **Datentypen erkennen:** Erstelle ein Array mit gemischten Werten. Filtere nur die Zahlen (`typeof`).
2. **Aggregation:** Implementiere `sumEvenNumbers()` mit `filter()` + `reduce()`.
3. **Objekt-Sortierung:** Sortiere ein Array von Objekten nach `preis` (auf- und absteigend).
4. **Suchfunktion:** Finde das erste verfügbare Produkt mit `find()`.
5. **Teilarray extrahieren:** Extrahiere die mittleren 3 Elemente mit `slice()`.
6. **DOM-Rendering vorbereiten:** Verwandle Strings in `<li>`-Elemente mit `map()`.
7. **Duplikatstruktur:** Verwende `flatMap()` für `[1, 2, 3] → [1,1,2,2,3,3]`.
8. **Eigene `join()`-Alternative:** Baue sie mit `reduce()`.
9. **Rahmung:** Entferne erstes und letztes Element via `slice(1, -1)`.
10. **Initialisierung:** Erstelle ein Array mit fünf `"?"` via `fill()`.

---

14. Micro-Projekt: Filterbare Produktliste (ergänzt)

Szenario: Eine dynamische Produktübersicht in einem Onlineshop soll:

- **nur verfügbare Artikel** anzeigen,
- nach Preis sortieren,
- die Artikel in einem lesbaren Format aufbereiten,
- **Gesamtpreis anzeigen**,
- im späteren Modul im DOM angezeigt werden können.

Datenstruktur:

```
const products = [
  { name: "Tisch", price: 120, available: true },
  { name: "Stuhl", price: 80, available: false },
  { name: "Lampe", price: 60, available: true }
];
```

Lösungen:

```
const available = products.filter(p => p.available);
const sorted = [...available].sort((a, b) => a.price - b.price);
const total = sorted.reduce((sum, p) => sum + p.price, 0);
const display = sorted.map(p => `${p.name} – ${p.price} €`).join("\n");
```

Ausgabe:

```
Lampe – 60 €
Tisch – 120 €
Gesamt: 180 €
```