JavaScript Arrays and Methods

An array in JavaScript is a special variable that can hold multiple values in a single variable. It is declared using square brackets `[]`.

Array Methods

1. **length**
   - Returns the number of elements in an array.
   - Example: `[1, 2, 3].length → 3`
2. **push(element)**
   - Adds an element to the end of an array.
   - Example: `let arr = [1, 2]; arr.push(3); → [1, 2, 3]`
3. **pop()**
   - Removes the last element from an array.
   - Example: `let arr = [1, 2, 3]; arr.pop(); → [1, 2]`
4. **unshift(element)**
   - Adds an element to the beginning of an array.
   - Example: `let arr = [2, 3]; arr.unshift(1); → [1, 2, 3]`
5. **shift()**
   - Removes the first element from an array.
   - Example: `let arr = [1, 2, 3]; arr.shift(); → [2, 3]`
6. **indexOf(element)**
   - Returns the first index of an element, or `-1` if not found.
   - Example: `["a", "b", "c"].indexOf("b") → 1`
7. **lastIndexOf(element)**
   - Returns the last index of an element, or `-1` if not found.
   - Example: `[1, 2, 3, 2].lastIndexOf(2) → 3`
8. **includes(element)**
   - Checks if an element exists in an array (returns `true` or `false`).
   - Example: `["a", "b", "c"].includes("b") → true`
9. **slice(start, end)**
   - Returns a new array from `start` to `end` (excluding `end`).
   - Example: `[1, 2, 3, 4].slice(1, 3) → [2, 3]`
10. **splice(start, deleteCount, item1, item2, ...)**
    - Removes/replaces elements in an array.
    - Example: `let arr = [1, 2, 3]; arr.splice(1, 1, 9); → [1, 9, 3]`
11. **concat(array1, array2, ...)**

- Merges two or more arrays.
- Example: `[1, 2].concat([3, 4])` → `[1, 2, 3, 4]`

12. **join(separator)**
    - Converts an array into a string with a given separator.
    - Example: `["a", "b", "c"].join("-")` → `"a-b-c"`
13. **reverse()**
    - Reverses the order of elements in an array.
    - Example: `[1, 2, 3].reverse()` → `[3, 2, 1]`
14. **sort()**
    - Sorts elements alphabetically (default).
    - Example: `["c", "a", "b"].sort()` → `["a", "b", "c"]`
15. **map(callbackFunction)**
    - Creates a new array by applying a function to each element.
    - Example: `[1, 2, 3].map(x => x * 2)` → `[2, 4, 6]`
16. **filter(callbackFunction)**
    - Returns elements that pass a condition.
    - Example: `[1, 2, 3, 4].filter(x => x % 2 === 0)` → `[2, 4]`
17. **reduce(callbackFunction, initialValue)**
    - Reduces array to a single value.
    - Example: `[1, 2, 3, 4].reduce((sum, x) => sum + x, 0)` → `10`
18. **every(callbackFunction)**
    - Checks if all elements satisfy a condition.
    - Example: `[2, 4, 6].every(x => x % 2 === 0)` → `true`
19. **some(callbackFunction)**
    - Checks if at least one element satisfies a condition.
    - Example: `[1, 2, 3].some(x => x % 2 === 0)` → `true`
20. **find(callbackFunction)**
    - Returns the first matching element.
    - Example: `[10, 20, 30].find(x => x > 15)` → `20`
21. **findIndex(callbackFunction)**
    - Returns the index of the first matching element.
    - Example: `[10, 20, 30].findIndex(x => x > 15)` → `1`
22. **fill(value, start, end)**
    - Fills elements with a value from `start` to `end`.
    - Example: `[1, 2, 3].fill(0, 1, 3)` → `[1, 0, 0]`
23. **flat(depth)**
    - Flattens nested arrays.

- Example: `[1, [2, [3]]].flat(2)` → `[1, 2, 3]`
- flat(2) method flattens the array two levels deep.

24. **flatMap(callbackFunction)**
    - Maps and flattens results.
    - Example: `[1, 2].flatMap(x => [x, x * 2])` → `[1, 2, 2, 4]`
    - .flatMap() flattens the result by one level only.
25. **at(index)**
    - Returns an element at a given index (supports negative indexes).
    - Example: `[10, 20, 30].at(-1)` → `30`
26. **toString()**
    - Converts an array to a string.
    - Example: `[1, 2, 3].toString()` → `'1,2,3'`
27. **toLocaleString()**
    - Converts an array to a localized string.
    - Example: `[1000].toLocaleString('en-US')` → `'1,000'`
28. **copyWithin(target, start, end)**
    - Copies part of an array within the same array.
    - Example: `[1, 2, 3, 4].copyWithin(1, 2, 4)` → `[1, 3, 4, 4]`
29. **keys()**
    - Returns an iterator of array keys.
    - Example: `let keys = ["a", "b"].keys(); console.log([...keys])` → `[0, 1]`
30. **values()**
    - Returns an iterator of array values.
    - Example: `let values = ["a", "b"].values(); console.log([...values])` → `["a", "b"]`
31. **entries()**
    - Returns an iterator of key-value pairs.
    - Example: `let entries = ["a", "b"].entries(); console.log([...entries])` → `[[0, "a"], [1, "b"]]`
32. **isArray(value)**
    - Checks if a value is an array.
    - Example: `Array.isArray([1, 2, 3])` → `true`

Mutating vs Non-Mutating Methods

- `filter`, `map`, `find`, `some`, `every`, and `reduce` methods **do not mutate** the original array. They return a **new array**.
- Methods like `push`, `pop`, `shift`, `unshift`, and `splice` **mutate** the original array.

Example: Using `filter()` Method

```
const items = [
  { name: "Item 1", price: 50 },
  { name: "Item 2", price: 150 },
  { name: "Item 3", price: 100 }
];

const filteredItems = items.filter((item) => item.price <= 100);

console.log(filteredItems);
// Output: [{ name: "Item 1", price: 50 }, { name: "Item 3", price: 100 }]

console.log(items);
// Original array remains unchanged:
// Output: [
//   { name: "Item 1", price: 50 },
//   { name: "Item 2", price: 150 },
//   { name: "Item 3", price: 100 }
// ]
```

Explanation:

- `filteredItems` has a **new memory address** since the `filter` method returns a **new array**.
- The **original `items` array remains unchanged**.

Conclusion

Arrays in JavaScript provide a wide variety of built-in methods that make manipulation easy. Understanding which methods **mutate** and which do not is key to writing efficient code.