Arrays

Array

Array are used to store multiple values in a single variable.

They are dynamic and can hold elements of different data types.

Creation

```
let teas = ["Masala Chai", "Green Chai"]; // Using array literals
let arr2 = new Array(5);   // Creates an empty array with length 5
let arr3 = Array.of(1, 2, 3); // Creates an array with the given elements
```

Accessing Elements

```
let teas = ["Masala Chai", "Green Chai", "Black Chai", "Ginger Chai"];

console.log(teas[0]); // First element - Masala Chai
console.log(teas[teas.length - 1]); // Last element - Ginger Chai
```

Array Methods

Adding Elements

```
let teas = ["Masala Chai", "Green Chai", "Black Chai", "Ginger Chai"];

teas.push("Lemon Chai"); // Add element to the end of the array
teas.unshift("Grey Chai"); // Add element to the beginning of the array
console.log(teas);
```

```
// Output: ["Grey Chai", "Masala Chai", "Green Chai", "Black Chai", "Ginger Chai",
```

Removing Elements

```
let teas = ["Masala Chai", "Green Chai", "Black Chai", "Ginger Chai"];

teas.pop(); // Remove the last element from the array
teas.shift(); // Remove the first element from the array
console.log(teas);

// Output: ["Masala Chai", "Green Chai", "Black Chai", "Ginger Chai"]
```

Modifying Elements

- Use `splice()` method → add, remove, or replace elements in an array.
- It modifies the original array and returns a new array of removed elements.

Syntax

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

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

// Remove 3 elements from index 1 to 3
const citrus = fruits.splice(1, 3); // splice is include end index

// Insert "Kiwi" at index 1
const addKiwi = fruits.splice(1, 0, "Kiwi");

console.log(citrus); // [ 'Orange', 'Lemon', 'Apple' ]
console.log(addKiwi); // []
console.log(fruits); //  [ 'Banana', 'Kiwi', 'Mango' ]
```

Extracting Elements

- Use `slice()` method → Extract a portion of an array without modifying the original array.
- `slice()` returns a new array and not changing the original array.

Syntax

```
array.slice(start[Optional], end[Optional])
```

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];

// slice is returning a smaller part of the original array
const citrus = fruits.slice(1, 3 + 1); // slice is not iclude end index

console.log(fruits); // [ 'Banana', 'Orange', 'Lemon', 'Apple', 'Mango' ]
console.log(citrus); // [ 'Orange', 'Lemon', 'Apple' ]
```

Searching Elements

Using `indexOf()`

- Returns the index of the first occurrence of the element, or `-1` if not found.
- Works for **primitive values** (numbers, strings, Booleans).

```
const arr = [10, 20, 30, 40, 50];
console.log(arr.indexOf(30));  // Output: 2
console.log(arr.indexOf(100)); // Output: -1
```

Using `find()`

- Advanced Searching Methods
- Finding Objects or Custom Conditions
- Returns the **first matching element** (or `undefined` if not found).
- Not return Array.

- It does not modify the original array.

Syntax

```
array.find(callback(element, index, array), thisArg);
```

```
const users = [
  { id: 1, name: "Amit" },
  { id: 2, name: "Neha" },
  { id: 3, name: "Rahul" }
];

const result = users.find(user => user.id === 2);
console.log(result); // Output: { id: 2, name: "Neha" }
```

Using `findIndex()`

- Advanced Searching Methods
- Returns the index of the **first matching element** (or `-1` if not found).

```
const users = [
  { id: 1, name: "Amit" },
  { id: 2, name: "Neha" },
  { id: 3, name: "Rahul" }
];

const index = users.findIndex(user => user.id === 2);
console.log(index); // Output: 1 - index
```

Using `filter()`

- filter returns a new array and not changing the original array with elements that satisfy a given condition.

Syntax

```
const newArray = array.filter(callback(element, index, array), thisArg);
```

```
const numbers = [10, 20, 30, 40, 50, 30];

const results = numbers.filter(num => num === 30);
console.log(results); // Output: [30, 30]
```

Checking Elements

Using `includes()`

- It returns `true` if the value is found, otherwise `false`.

Syntax

```
array.includes(value, startIndex[Optional]);
string.includes(value, startIndex[Optional]);
```

```
const arr = [10, 20, 30, 40, 50];
console.log(arr.includes(30));  // Output: true
console.log(arr.includes(100)); // Output: false
```

Using `some()` and `every()`

- `some()` → Returns `true` if **at least one** element matches.
- `every()` → Returns `true` if **all** elements match.

Syntax

```
array.some(callback(element, index, array), thisArg);
```

```
array.every(callback(element, index, array), thisArg);
```

```
const numbers = [10, 20, 30, 40, 50];

console.log(numbers.some(num ⇒ num > 25));  // Output: true
console.log(numbers.every(num ⇒ num > 25)); // Output: false
```

Iterating Over an Array

Using `forEach()`

- A higher-order function that executes a callback function for each element.
- It is modifies the original array.
- `forEach()` does not return anything (undefined).
- ❌ Cons: Cannot use `break` or `continue`

Syntax

```
array.forEach(callback(element, index(optional), array(optional)), thisArg(optiona
```

```
let teas = ["Masala Chai", "Green Chai", "Black Chai", "Ginger Chai"];

teas.forEach((chai, index) ⇒ {
  console.log(`${index}: ${chai}`);
}) // chai - all values of arr

// Output
// 0: Masala Chai
// 1: Green Chai
// 2: Black Chai
// 3: Ginger Chai
```

Using `map()`

- `map()` does not modify the original array.
- `forEach` vs `map` - `map` returns a new array but `forEach` doesn't.

- ❌ Cons: Cannot break early
- ✅ Pros: Returns a new array

Syntax

```
array.map(callback(element, index, array), thisArg);
```

```
let numbers = [1, 2, 3, 4, 5];

let newArr = numbers.map(value => value * 2);
console.log(newArr); // [2, 4, 6, 8, 10]
```

Using `reduce()`

- Reduces an array to a single value.
- `reduce()` returns a single value and not changing the original array.
- It is commonly used for summation, product calculation, flattening arrays, and more

Syntax

```
array.reduce(callback, initialValue[Optional])
```

```
let numbers = [1, 2, 3, 4, 5];

let newArr = numbers.map(value => value * 2);
console.log(newArr); // [2, 4, 6, 8, 10]

let sum = newArr.reduce((total, value) => total + value);
console.log(sum); // 30
```

Using `filter()`

- Filters elements based on a condition.
- `filter()` returns a new array and not changing the original array.

Syntax

```
const newArray = array.filter(callback(element, index(optional), array(optional)), t
```

```
let numbers = [10, 20, 30, 40, 50];

let evenNumbers = numbers.filter(value ⇒ value % 2 === 0);
console.log(evenNumbers); // [10, 20, 30, 40]
```

Sorting

Using `.sort()`

- Default sorting works for strings but not for numbers.
- modifying the original array.
- Use a compare function to correctly sort numbers.

Syntax

```
array.sort(compareFunction(optional));
// compareFunction (optional) → Defines the sorting order.
```

1. Default Sorting (Lexicographic Order)

```
const fruits = ["banana", "apple", "cherry"];
console.log(fruits.sort()); // Output: ["apple", "banana", "cherry"]

const numbers = [10, 5, 30, 2];
console.log(numbers.sort()); // Output: [10, 2, 30, 5] ❌ (incorrect for numbers)
```

2. Sorting Numbers (Using a Compare Function)

```
// To correctly sort numbers, use a compare function:

const numbers = [10, 5, 30, 2];
```

```
// Ascending order (small to large)
numbers.sort((a, b) => a - b);
console.log(numbers); // Output: [2, 5, 10, 30]

// Descending order (large to small)
numbers.sort((a, b) => b - a);
console.log(numbers); // Output: [30, 10, 5, 2]
```

3. Sorting Objects (by Property)

```
// Useful when sorting an array of objects:

const users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 22 },
  { name: "Charlie", age: 30 },
];

// Sort by age (ascending)
users.sort((a, b) => a.age - b.age);
console.log(users);

/* Output:
[
  { name: "Bob", age: 22 },
  { name: "Alice", age: 25 },
  { name: "Charlie", age: 30 }
]
*/
```

4. Sorting Strings with Different Cases

```
// Use localeCompare() to handle case-insensitive sorting:
const names = ["Zebra", "apple", "Orange", "banana"];
```

```
names.sort((a, b) => a.localeCompare(b, "en", { sensitivity: "base" }));
console.log(names); // Output: ["apple", "banana", "Orange", "Zebra"]
```

Reversing an Array

Using `.reverse()`

- Built-in Method
- `reverse()` **modifies** the original array.

```
let originalArr = [1, 2, 3, 4, 5];
let reversedArr = [...originalArr].reverse();
console.log(reversedArr);
// Output: [5, 4, 3, 2, 1]
```

- If you want to **avoid modifying** the original array

```
let originalArr = [1, 2, 3, 4, 5];
let reversedArr = [...originalArr].reverse();

console.log(reversedArr); // [5, 4, 3, 2, 1]
console.log(originalArr); // [1, 2, 3, 4, 5]
```

Concatenation of Array

Using `concat()`

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4, 5, 6]
```

Using Spread Operator (`...`)

```
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const result = [...arr1, ...arr2];
console.log(result); // [1, 2, 3, 4, 5, 6]
```

`join()` Method

- Convert an array into a string.

Syntax

```
array.join(separator(optional));
```

1. Default Behavior (`,` as separator)

```
const arr = [1, 2, 3, 4, 5];
console.log(arr.join()); // "1,2,3,4,5"
```

2. Custom Separator

```
const words = ["JavaScript", "is", "awesome"]; // Recommended
console.log(words.join(" ")); // "JavaScript is awesome"


const arr = ["Apple", "Banana", "Cherry"];
console.log(arr.join(" - ")); // "Apple - Banana - Cherry"
```

3. Without a Separator (Empty String)

```
const arr = ["H", "E", "L", "L", "O"];
console.log(arr.join("")); // "HELLO"
```

4. Joining Numbers

```
const arr = [10, 20, 30];
console.log(arr.join(" | ")); // "10 | 20 | 30"
```

5. Joining Nested Arrays

```
const arr = [[1, 2], [3, 4], [5, 6]];
console.log(arr.join("-")); // "1,2-3,4-5,6"
```