ARRAY **MAP** METHOD IN JAVASCRIPT

ARRAY MAP METHOD

The `map()` method of an array can be used to transform the elements of an array by applying a callback function to each element of the source array. It returns a new array with the transformed data.

- 👉 Takes a source array on which it works.
- 👉 Applies a function (callback) to each element of the array
- 👉 Returns a new array with the transformed data.
- 👉 **Important:** It does NOT modify the original array.

ARRAY MAP METHOD

The `map()` method of an array can be used to transform the elements of an array by applying a callback function to each element of the source array. It returns a new array with the transformed data.

```
//Squaring the elements of an array

const numbers = [1, 4, 9];
const squares = numbers.map(num => num * num); // [1, 16, 81]

console.log(numbers); // [1, 4, 9] (original remains unchanged)
console.log(squares);  // [1, 16, 81] (new array with squares)
```

ARRAY MAP METHOD

The `map()` method of an array can be used to transform the elements of an array by applying a callback function to each element of the source array. It returns a new array with the transformed data.

```
//Converting string elements of the array to Upper Case

const names = ["apple", "banana", "cherry"];
const upperNames = names.map(name => name.toUpperCase());

console.log(names);  // ["apple", "banana", "cherry"]
console.log(upperNames); // ["APPLE", "BANANA", "CHERRY"]
```

ARRAY MAP METHOD

The `map()` method of an array can be used to transform the elements of an array by applying a callback function to each element of the source array. It returns a new array with the transformed data.

```
//Extracting specified properties from an array of objects
const products = [
  { id: 1, name: "Shirt", price: 20 },
  { id: 2, name: "Hat", price: 15 },
];

const productNames = products.map(product => product.name); // ["Shirt",
"Hat"]

console.log(productNames); // ["Shirt", "Hat"]
```

TRANSFORMATION ARRAY METHODS

There are three transformation methods in JavaScript which can be applied on an array:

- 👉 `map()`
- 👉 `filter()`
- 👉 `reduce()`

ARRAY FILTER METHOD IN JAVASCRIPT

ARRAY FILTER METHOD

The **`filter()`** method in JavaScript is a powerful tool for creating a new array by filtering elements from an existing array based on a specified condition. It iterates over each element of the original array and applies a callback function to it. If the callback function returns true for a particular element, that element is included in the new array. Otherwise, it's excluded.

👉 Takes a source array on which it works.

👉 Applies a function (callback) to each element of the array

👉 Returns a new array with the filtered elements.

👉 **Important:** It does NOT modify the original array.

👉 You can chain multiple **`filter()`** calls to apply multiple filtering conditions.

ARRAY FILTER METHOD

The **`filter()`** method in JavaScript is a powerful tool for creating a new array by filtering elements from an existing array based on a specified condition. It iterates over each element of the original array and applies a callback function to it. If the callback function returns true for a particular element, that element is included in the new array. Otherwise, it's excluded.

```
//Filter all the even numbers & return a new array with filtered elements

const numbers = [1, 2, 3, 4, 5, 6];

const evenNumbers = numbers.filter(number => number % 2 === 0);
console.log(evenNumbers); // Output: [2, 4, 6]
```

ARRAY FILTER METHOD

The **`filter()`** method in JavaScript is a powerful tool for creating a new array by filtering elements from an existing array based on a specified condition. It iterates over each element of the original array and applies a callback function to it. If the callback function returns true for a particular element, that element is included in the new array. Otherwise, it's excluded.

```
//Filter all the elements with number of characters greater than 5 & return a
new array with filtered elements

const words = ['apple', 'banana', 'cherry', 'date', 'elderberry’];

const longWords = words.filter(word => word.length > 5);
console.log(longWords); // Output: ['banana', 'elderberry']
```

ARRAY FILTER METHOD

The **`filter()`** method in JavaScript is a powerful tool for creating a new array by filtering elements from an existing array based on a specified condition. It iterates over each element of the original array and applies a callback function to it. If the callback function returns true for a particular element, that element is included in the new array. Otherwise, it's excluded.

```javascript
//Filter all the object with age greater than 18 & return a new array with filtered elements
const people = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 22 }
];

const adults = people.filter(person => person.age >= 18);
console.log(adults);
// Output: [
//  { name: 'Alice', age: 25 },
//  { name: 'Bob', age: 30 }, { name: 'Charlie', age: 22 }
//]
```

ARRAY REDUCE METHOD IN JAVASCRIPT

ARRAY REDUCE METHOD

The **`reduce()`** method is a powerful tool in JavaScript that allows you to iterate over an array and accumulate a single value. It takes a callback function as an argument and applies it to each element of the array, reducing it to a single value.

👉 Takes a source array on which it works.

👉 Takes two arguments – A callback function & an accumulator

👉 The callback function is executed for each element of the array.

👉 The return value of the callback function becomes the new accumulator for the next iteration.

👉 Returns a single value **not an array.**

ARRAY REDUCE METHOD

The **`reduce()`** method is a powerful tool in JavaScript that allows you to iterate over an array and accumulate a single value. It takes a callback function as an argument and applies it to each element of the array, reducing it to a single value.

```
//Returns the sum of all elements of array
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 0); // Initial value is 0

console.log(sum); // Output: 15
```

ARRAY REDUCE METHOD

The **`reduce()`** method is a powerful tool in JavaScript that allows you to iterate over an array and accumulate a single value. It takes a callback function as an argument and applies it to each element of the array, reducing it to a single value.

```
//Returns the sum of all elements of array
const numbers = [10, 5, 20, 8, 15];

const max = numbers.reduce((accumulator, currentValue) => {
  return Math.max(accumulator, currentValue);
}); // No initial value, first element is used

console.log(max); // Output: 20
```

ARRAY FOREACH METHOD IN JAVASCRIPT

ARRAY FOREACH METHOD

The **`forEach()`** method is a powerful tool in JavaScript for iterating over elements of an array. It allows you to execute a provided function for each element in the array.

👉 Takes a source array on which it works.

👉 Takes a callback function as the argument

👉 The code within the callback function is executed for each element, allowing you to perform operations on each element as needed.

👉 Return type for **`forEach()`** is void.

ARRAY FOREACH METHOD

The **`forEach()`** method is a powerful tool in JavaScript for iterating over elements of an array. It allows you to execute a provided function for each element in the array.

```
//Loop over each element of numbers array and log a message in the developer
console.

const numbers = [1, 2, 3, 4, 5];

numbers.forEach(function(number, index) {
  console.log(`Number at index ${index}: ${number}`);
});
```

SLICE & SPLICE
ARRAY METHODS

SLICE METHOD

The **`slice()`** method creates a new array by extracting a portion of an existing array. It doesn't modify the original array.

👉 Takes a source array on which it works.

👉 Takes two arguments, the index from where it should extract & the number of elements it should extract.

👉 Returns a new array.

👉 Does not modify the original array.

SLICE METHOD

The **`slice()`** method creates a new array by extracting a portion of an existing array. It doesn't modify the original array.

```
const fruits = ["apple", "banana", "cherry", "date", "elderberry"];

// Extract elements from index 1 (inclusive) to 3 (exclusive)
const citrusFruits = fruits.slice(1, 3);
console.log(citrusFruits); // Output: ["banana", "cherry"]

// Extract elements from index 2 to the end
const laterFruits = fruits.slice(2);
console.log(laterFruits); // Output: ["cherry", "date", "elderberry"]
```

SPLICE METHOD

The **`splice()`** method modifies an array by removing, replacing, or adding elements. It directly manipulates the original array.

👉 Takes a source array on which it works.

👉 Takes two arguments, the index from where it should extract & the number of elements

it should extract.

👉 It modifies the original array.

SPLICE METHOD

The **`splice()`** method modifies an array by removing, replacing, or adding elements. It directly manipulates the original array.

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

// Remove 2 elements starting from index 2
numbers.splice(2, 2);
console.log(numbers); // Output: [1, 2, 5]

// Remove 1 element starting from index 1 and insert "a" and "b"
numbers.splice(1, 1, "a", "b");
console.log(numbers); // Output: [1, "a", "b", 5]
```

SUBSTRING & SUBSTR

STRING METHODS

SUBSTRING METHOD

The **`substring()`** method creates a new string by extracting a portion of an existing string. It doesn't modify the original string value.

- 👉 Takes a source string on which it works.
- 👉 Takes two arguments, the start index & end index .
- 👉 Returns a new string.
- 👉 Does not modify the original string.

SUBSTRING METHOD

The **`substring()`** method creates a new string by extracting a portion of an existing string. It doesn't modify the original string value.

```
//The substring method will return a new string value with extracted
character from a given string.

let str = "Hello, World!";

let substr1 = str.substring(7); // "World!"
let substr2 = str.substring(0, 5); // "Hello"
```

SUBSTR METHOD

The **`substr()`** method creates a new string by extracting a portion of an existing string. It doesn't modify the original string value.

👉 Takes a source string on which it works.

👉 Takes two arguments, the start index from where it should extract & the number of characters it should extract.

👉 Returns a new string.

👉 Does not modify the original string.

SUBSTR METHOD

The **`substr()`** method creates a new string by extracting a portion of an existing string. It doesn't modify the original string value.

```
//The substr method will return a new string value with extracted character
from a given string.

let str = "Hello, World!";

let substr1 = str.substr(7); // "World!"
let substr2 = str.substr(0, 5); // "Hello"
```

DIFFERENCE BETWEEN SUBSTRING & SUBSTR

To **`substring()`** method when specified a negative value, it considers it as index. On the other hand, **`substr()`** method considers it as length.

```
let str = "Hello, World!";

let substr1 = str.substring(7); // "World!"
let substr2 = str.substring(-5); // "World!" (same as str.substring(7))

let substr1 = str.substr(7); // "World!"
let substr2 = str.substr(-5); // "orld!" (starts from the 5th character from
the end)
```

SPLIT & JOIN METHODS

SPLIT METHOD

The **`split()`** method is used to split a string into an array of substrings based on a specified separator.

👉 Takes a source string on which it works.

👉 Takes a separator character as an argument.

👉 Returns a new array with string elements.

Parameters:

👉 **separator** (optional): A string or regular expression that specifies the separator. If omitted, the entire string is treated as a single element.

👉 **limit** (optional): A number specifying the maximum number of splits.

SPLIT METHOD

The **`split()`** method is used to split a string into an array of substrings based on a specified separator.

```
const str = "Hello, World!";

const words = str.split(" "); // ["Hello,", "World!"]
const chars = str.split(""); // ["H", "e", "l", "l", "o", ",", " ", "W", "o",
"r", "l", "d", "!"]
```

JOIN METHOD

The **`join()`** method is used to join the elements of an array into a string, using a specified separator.

👉 Takes a source array on which it works.

👉 Takes a separator character as an argument.

👉 Returns a new string value separated by the separator.

Parameters:

👉 **separator** (optional): A string to be used as a separator. If omitted, the default separator is a comma (**,**).

JOIN METHOD

The **`join()`** method is used to join the elements of an array into a string, using a specified separator.

```
const words = ["Hello", "World"];

const sentence = words.join(" "); // "Hello World"
const commaSeparated = words.join(","); // "Hello,World"
```

SORT & REVERSE
ARRAY METHODS

ARRAY SORT METHOD

The **`sort()`** method sorts the elements of an array in place and returns the sorted array. By default, **`sort()`** converts the elements to strings and sorts them in alphabetical order. This can lead to unexpected results when sorting numbers.

👉 The **`sort()`** method modifies the original array directly.

👉 The **`sort()`** method returns an array by modifying the original array.

```
//Sorting string elements of the array

const names = [“steve", “john", “merry"];
const sortedNames = names.sort()

console.log(names);  // [“john", “merry", “steve"]
console.log(sortedNames); // [“john", “merry", “steve"]
```

ARRAY SORT METHOD

To sort numbers correctly, you need to provide a comparison function to the **`sort()`** method. The provided comparison function `((a, b) => a - b)` determines the sort order:

👉 If `a - b` is negative, **a** comes before **b**.
👉 If `a - b` is positive, **b** comes before **a**.
👉 If `a - b` is zero, **a** and **b** remain in their original order.

```
//Sorting string elements of the array

const names = [“steve", “john", “merry"];
const sortedNames = names.sort()

console.log(names);  // [“john", “merry", “steve"]
console.log(sortedNames); // [“john", “merry", “steve"]
```

ARRAY SORT METHOD

You can sort arrays of objects by specifying a property to compare. In the following example. we are sorting the objects based on a Number property.

```
const users = [
  { name: "Alice", age: 25 },
  { name: "Bob", age: 30 },
  { name: "Charlie", age: 20 },
];

users.sort((a, b) => {
  if (a.age < b.age) return -1;
  if (a.age > b.age) return 1;
  return 0;
});

console.log(users);
```

ARRAY FIND METHOD IN JAVASCRIPT

ARRAY FIND METHOD

The **`find()`** method iterates over an array and returns the first matching element from the array that satisfies a given condition provided in callback function.

👉 Takes a source array on which it works.

👉 Applies a function (callback) to each element of the array. From this callback function,

you specify a condition which returns a **Boolean** value.

👉 Returns the first matching element from the array which satisfies the given condition.

Else, it returns `undefined.`

👉 **Important:** It does NOT modify the original array.

ARRAY FIND METHOD

The **`find()`** method iterates over an array and returns the first matching element from the array that satisfies a given condition provided in callback function.

```
const numbers = [5, 12, 8, 130, 44];

const found = numbers.find(element => element > 10);

console.log(found); // Output: 12
```

ARRAY FIND METHOD

The **`find()`** method iterates over an array and returns the first matching element from the array that satisfies a given condition provided in callback function.

```
const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
  { id: 3, name: "Charlie" },
];

const foundUser = users.find(user => user.id === 2);

console.log(foundUser);
// Output: { id: 2, name: "Bob" }
```

FINDINDEX METHOD IN JAVASCRIPT

ARRAY FINDINDEX METHOD

The **`findIndex()`** method iterates over an array and returns the index of first matching element from the array that satisfies a given condition provided in callback function.

👉 Takes a source array on which it works.

👉 Applies a function (callback) to each element of the array. From this callback function, you specify a condition which returns a **Boolean** value.

👉 Returns the index of first matching element from the array which satisfies the given condition. Else, it returns `-1.`

👉 **Important:** It does NOT modify the original array.

ARRAY FINDINDEX METHOD

The **`findIndex()`** method iterates over an array and returns the index of first matching element from the array that satisfies a given condition provided in callback function.

```
const numbers = [5, 12, 8, 130, 44];

const foundIndex = numbers.findIndex(element => element > 10);

console.log(foundIndex); // Output: 1
```

ARRAY FINDINDEX METHOD

The **`findIndex()`** method iterates over an array and returns the index of first matching element from the array that satisfies a given condition provided in callback function.

```
const users = [
  { id: 1, name: "Alice" },
  { id: 2, name: "Bob" },
  { id: 3, name: "Charlie" },
];

const foundIndex = users.findIndex(user => user.id === 2);

console.log(foundUser);
// Output: 1
```

SOME & EVERY
ARRAY METHODS

ARRAY SOME METHOD

The `some()` method tests whether at least one element in the array passes the test implemented by the provided function.

Return Value:

👉 If at least one element passes the test, `some()` returns `true`.
👉 If none of the elements pass the test, `some()` returns `false`.

Key Points:

👉 The `some()` method only checks if at least one element satisfies the condition, not all elements.
👉 The `some()` method does not modify the original array.

ARRAY SOME METHOD

The **`some()`** method tests whether at least one element in the array passes the test implemented by the provided function.

```
const numbers = [5, 12, 8, 130, 44];

const hasEvenNumbers = numbers.some(number => number % 2 === 0);

console.log(hasEvenNumbers); // Output: true
```

ARRAY SOME METHOD

The **`some()`** method tests whether at least one element in the array passes the test implemented by the provided function.

```
const users = [
  { id: 1, name: "Alice“, isActive: true },
  { id: 2, name: "Bob", isActive: false  },
  { id: 3, name: "Charlie", isActive: true  },
];

const hasActiveUsers = users.some(user => user.isActive);

console.log(hasActiveUsers);
// Output: true
```

ARRAY EVERY METHOD

The **`every()`** method tests whether all the element in the array passes the test implemented by the provided callback function.

Return Value:

👉 If all elements passes the test, **`every()`** returns `true`.

👉 If any element fails the test, **`every()`** returns `false`.

Key Points:

👉 The **`every()`** method checks if all the elements satisfies the condition, not just one element.

👉 The **`every()`** method does not modify the original array.

ARRAY EVERY METHOD

The **`every()`** method tests whether all the element in the array passes the test implemented by the provided callback function.

```
const numbers = [2, 4, 8, 16, 28];

const allEven = numbers.some(number => number % 2 === 0);

console.log(allEven); // Output: true
```

ARRAY EVERY METHOD

The **every()** method tests whether all the element in the array passes the test implemented by the provided callback function.

```
const users = [
  { id: 1, name: "Alice“, isActive: true },
  { id: 2, name: "Bob", isActive: false  },
  { id: 3, name: "Charlie", isActive: true  },
];

const allActiveUsers = users.some(user => user.isActive);

console.log(allActiveUsers);
// Output: false
```

FLAT & FLATMAP ARRAY METHODS

ARRAY FLAT METHOD

The `flat()` method creates a new array by recursively flattening a given array up to a specified depth. If no depth is specified, it defaults to **1**.

👉 **`depth` (optional):** Specifies how deep a nested array structure should be flattened. Defaults to 1.

👉 `Infinity` can be used to flatten all nested arrays regardless of their depth.

👉 Returns a new array without modifying the original array.

ARRAY FLAT METHOD

The `flat()` method creates a new array by recursively flattening a given array up to a specified depth. If no depth is specified, it defaults to **1**.

Example 1: Flattening a single level

```
const nestedArray = [1, 2, [3, 4]];

const flattenedArray = nestedArray.flat();

console.log(flattenedArray); // Output: [1, 2, 3, 4]
```

ARRAY FLAT METHOD

The `flat()` method creates a new array by recursively flattening a given array up to a specified depth. If no depth is specified, it defaults to **1**.

Example 2: Flattening multiple levels

```
const deeplyNestedArray = [1, 2, [3, 4, [5, 6]]];
const flattenedOnce = deeplyNestedArray.flat();
console.log(flattenedOnce); // Output: [1, 2, 3, 4, [5, 6]]

const flattenedTwice = deeplyNestedArray.flat(2); // Flatten to a depth of 2
console.log(flattenedTwice); // Output: [1, 2, 3, 4, 5, 6]

const flattenedAll = deeplyNestedArray.flat(Infinity); // Flatten all levels
console.log(flattenedAll); // Output: [1, 2, 3, 4, 5, 6]
```

ARRAY FLAT METHOD

The `flat()` method creates a new array by recursively flattening a given array up to a specified depth. If no depth is specified, it defaults to **1**.

Example 3: Remove empty slots in sparse arrays

```
const sparseArray = [1, , 3, [4, , 6]]; // Note the empty slots

const flattenedSparse = sparseArray.flat();

console.log(flattenedSparse); // Output: [1, 3, 4, 6]
```

ARRAY FLATMAP METHOD

The `flatMap()` method first maps each element of the array to a new array using a provided function, and then flattens the result into a new array. It's effectively a combination of `map()` and `flat()` (with a depth of 1).

👉 Returns a new array without modifying the original array.

ARRAY FLATMAP METHOD

The `flatMap()` method first maps each element of the array to a new array using a provided function, and then flattens the result into a new array. It's effectively a combination of `map()` and `flat()` (with a depth of 1).

Example 4: Get all skills using `map()` & `flat()` method

```
const employees = [
  {id: 1, name: ‘john’, skills: [‘HTML’, ‘CSS’, ‘JAVASCRIPT’]},
  {id: 2, name: ‘mark’, skills: [‘C#’, ‘SQL’]},
  {id: 3, name: ‘merry’, skills: [‘Angular’, ‘React’]},
]

const skills = employees.map((emp) => emp.skills).flat();

console.log(skills); // [‘HTML’, ‘CSS’, ‘JAVASCRIPT’, ‘C#’, ‘SQL’, ‘Angular’,
‘React’]
```

ARRAY FLATMAP METHOD

The `flatMap()` method first maps each element of the array to a new array using a provided function, and then flattens the result into a new array. It's effectively a combination of `map()` and `flat()` (with a depth of 1).

Example 5: Get all skills using `flatMap()` method

```
const employees = {
  {id: 1, name: ‘john’, skills: [‘HTML’, ‘CSS’, ‘JAVASCRIPT’]},
  {id: 2, name: ‘mark’, skills: [‘C#’, ‘SQL’]},
  {id: 3, name: ‘merry’, skills: [‘Angular’, ‘React’]},
}

const skills = employees.flatMap((emp) => emp.skills);

console.log(skills); // [‘HTML’, ‘CSS’, ‘JAVASCRIPT’, ‘C#’, ‘SQL’, ‘Angular’,
‘React’]
```

THE TOUPPERCASE & TOLOWERCASE METHOD

TOUPPERCASE METHOD ON STRING

The `toUpperCase()` method creates and returns a new string that is the uppercase version of the original string. The original string remains unchanged.

```
const str = "hello world";
const upperStr = str.toUpperCase();

console.log(str);       // Output: hello world (original string unchanged)
console.log(upperStr);    // Output: HELLO WORLD (new uppercase string)

const mixedCase = "MiXeD CaSe";
const allCaps = mixedCase.toUpperCase();
console.log(allCaps); // Output: MIXED CASE
```

TOLOWERCASE METHOD ON STRING

The `toLowerCase()` method works in the opposite way. It creates and returns a new string that is the lowercase version of the original string. Again, the original string is not modified.

```
const str = "HELLO WORLD";
const lowerStr = str.toLowerCase();

console.log(str);       // Output: HELLO WORLD (original string unchanged)
console.log(lowerStr);    // Output: hello world (new lowercase string)

const mixedCase = "MiXeD CaSe";
const allLower = mixedCase.toLowerCase();
console.log(allLower); // Output: mixed case
```

ES2022 AT METHOD IN JAVASCRIPT

ES2022 AT METHOD

The `at()` method in JavaScript is a relatively new addition (ES2022) to arrays and strings that provides a concise way to access elements at a specific index, including negative indices. It's designed to simplify accessing elements from the end of an array or string.

ES2022 AT METHOD

The `at()` method in JavaScript is a relatively new addition (ES2022) to arrays and strings that provides a concise way to access elements at a specific index, including negative indices. It's designed to simplify accessing elements from the end of an array or string.

```
const myArray = ['apple', 'banana', 'cherry', 'date'];

console.log(myArray.at(0));   // Output: apple
console.log(myArray.at(2));   // Output: cherry
console.log(myArray.at(-1));  // Output: date
console.log(myArray.at(-2));  // Output: cherry

// Equivalent using bracket notation (more verbose for negative indices)
console.log(myArray[myArray.length - 1]); // Output: date
console.log(myArray[myArray.length - 2]); // Output: cherry

// Out-of-bounds indices:
console.log(myArray.at(5));   // Output: undefined
console.log(myArray.at(-5));  // Output: undefined
```

ES2022 AT METHOD

The `at()` method in JavaScript is a relatively new addition (ES2022) to arrays and strings that provides a concise way to access elements at a specific index, including negative indices. It's designed to simplify accessing elements from the end of an array or string.

```
const myString = "hello";

console.log(myString.at(0));   // Output: h
console.log(myString.at(2));   // Output: l
console.log(myString.at(-1));  // Output: o
console.log(myString.at(-2));  // Output: l

// Out-of-bounds indices:
console.log(myString.at(5));   // Output: undefined
console.log(myString.at(-6));   // Output: undefined
```

THE INDEXOF & LASTINDEXOF METHOD

STRING IN JAVASCRIPT

A string in JavaScript is a sequence of characters. Each character in a string is positioned at an index. The index in a string start from `0`.

INDEXOF METHOD

The `indexOf()` method returns the first index at which a given element can be found in the string or array, or `-1` if it is not present.

```
const str = "Hello world, hello!";

console.log(str.indexOf("hello"));    // Output: 0 (first occurrence)
console.log(str.indexOf("world"));    // Output: 6
console.log(str.indexOf("hello", 6)); // Output: 13 (starts search from index
6)
console.log(str.indexOf("xyz"));      // Output: -1 (not found)
```

INDEXOF METHOD

The `indexOf()` method returns the first index at which a given element can be found in the string or array, or `-1` if it is not present.

```
const arr = [1, 2, 3, 2, 4, 2];

console.log(arr.indexOf(2));        // Output: 1 (first occurrence)
console.log(arr.indexOf(2, 3));     // Output: 5 (starts search from index 3)
console.log(arr.indexOf(5));        // Output: -1 (not found)
```

LASTINDEXOF METHOD

The `lastIndexOf()` method returns the last index at which a given element can be found in the string or array, or `-1` if it is not present. It searches backward from the end.

```
const str = "Hello world, hello!";

console.log(str.lastIndexOf("hello"));    // Output: 13 (last occurrence)
console.log(str.lastIndexOf("world"));    // Output: 6
console.log(str.lastIndexOf("hello", 10)); // Output: 0 (searches backward
from index 10)
console.log(str.lastIndexOf("xyz"));      // Output: -1 (not found)
```

LASTINDEXOF METHOD

The `lastIndexOf()` method returns the last index at which a given element can be found in the string or array, or `-1` if it is not present. It searches backward from the end.

```
const arr = [1, 2, 3, 2, 4, 2];

console.log(arr.lastIndexOf(2));        // Output: 5 (last occurrence)
console.log(arr.lastIndexOf(2, 4));     // Output: 3 (searches backward from
index 4)
console.log(arr.lastIndexOf(5));        // Output: -1 (not found)
```

THE SLICE METHOD OF STRING

SLICE METHOD ON STRING

We can also use the `slice()` method on strings in JavaScript. Strings are primitive values, but when you call a method like `slice()` on them, JavaScript temporarily converts them to String objects to perform the operation.

SLICE VS SUBSTRING METHOD

Both `slice()` and `substring()` can be used to extract parts of a string, but they have some key differences:

Handling of Negative Indices:

👉 `slice()`: Accepts negative indices. Negative indices count from the end of the string.

👉 `substring()`: Does not handle negative indices correctly. It treats negative indices as `0`.

Handling of startIndex greater than endIndex:

👉 `slice()`: Returns an empty string if **startIndex** is greater than **endIndex**.

👉 `substring()`: Swaps the **startIndex** and **endIndex** if **startIndex** is greater than **endIndex**. It effectively treats the smaller index as the start and the larger index as the end.

LENGTH PROPERTY

The `length` property of a string in JavaScript returns the number of characters in that string. It's a fundamental property that you'll use very frequently when working with strings.

When used on an array, the `length` property returns the number of elements in the array.

THE TRIM METHODS OF STRING

TRIM METHODS IN JAVASCRIPT

There are three in javaScript which can be used to trim (remove) spaces from start or end of a string. These methods are:

👉 `trimStart()`: Removes white spaces from the start of the string.

👉 `trimEnd()`: Removes white spaces from the end of the string.

👉 `trim()`: Removes white spaces from both start & end of the string.

THE REPLACE METHOD OF STRING

REPLACE METHOD IN JAVASCRIPT

The `replace()` method in JavaScript is a powerful tool for manipulating strings. It allows you to find specific patterns within a string and replace them with new content.

The `replace()` searches a string for a specified value (which can be a string or a regular expression) and returns a new string where the first (or all, if using a regular expression with the global flag) occurrences of the matched pattern are replaced with a specified replacement value. The original string remains unchanged.

REPLACE METHOD IN JAVASCRIPT

Example 1: Simple String Replacement

The `replace()` method can be used to replace a substring inside a given string with another substring. This is the most basic use case. In following example, it replaces the first occurrence of "world" with "JavaScript".

```
//Replace the first occurrence of word "world" with "JavaScript"
const message = "Hello, world!";

const newMessage = message.replace("world", "JavaScript");
console.log(newMessage);

// Output: Hello, JavaScript!
```

REPLACE METHOD IN JAVASCRIPT

Example 2: Replacing All Occurrences (with Regular Expression and g flag)

In following example, the **`replace()`** method replaces all the occurrence of word “test" with “experiment".

```
//Replace all the occurrence of word “test" with “experiment"
const text = "This is a test. This is another test.";

const newText = text.replace(/test/g, "experiment");
console.log(newText);

// Output: This is an experiment. This is another experiment.
```

REPLACE METHOD IN JAVASCRIPT

Example 3: Case-Insensitive Replacement (with Regular Expression and i flag)

By default, replace method is case sensitive. If you want to do case insensitive replacement you can use regular expression with `i` flag.

```
// The i (case-insensitive) flag makes the search for "hello" case-
insensitive.

const greeting = "Hello there!";

const newGreeting = greeting.replace(/hello/i, "Hi");
console.log(newGreeting); // Output: Hi there!
```

REPLACE METHOD IN JAVASCRIPT

Example 4: Replacing with a Function (Dynamic Replacement)

This is where **`replace()`** becomes very powerful. You can provide a function as the second argument (newValue). This function will be called for each match, and its return value will be used as the replacement.

```
const numbers = "1 2 3 4 5";

const doubledNumbers = numbers.replace(/\d+/g, (match) => {
    const num = parseInt(match);
    return num * 2;
});
console.log(doubledNumbers); // Output: 2 4 6 8 10
```

REPLACE METHOD IN JAVASCRIPT

Example 5: Using Captured Groups in Regular Expressions

You can use parentheses () in your regular expression to create captured groups. These groups can then be referenced in the replacement string (or in the replacement function) using **$1**, **$2**, etc.

```
// Below example swaps the first and last names.

const name = "Doe, John";

const formattedName = name.replace(/(\w+), (\w+)/, "$2 $1");
console.log(formattedName); // Output: John Doe
```

REPLACE METHOD IN JAVASCRIPT

Example 6: Replacing Special Characters:

If you need to replace special characters that have meaning in regular expressions **(like `.`, `*`, `+`, `?`, etc.)**, you need to escape them using a backslash \.

```
// Replace all periods with exclamation marks

const text = "This is a test.";

const newText = text.replace(/\./g, "!");

console.log(newText); // Output: This is a test!
```

THE INCLUDES, STARTSWITH & ENDSWITH METHOD

INCLUDES, STARTSWITH & ENDSWITH METHOD

The `includes()`, `startsWith()`, and `endsWith()` are string methods in JavaScript that provide convenient ways to check if a string contains a specific substring or if it starts or ends with a particular string. They are generally simpler and more readable than using regular expressions for these specific purposes.

👉 All these three method returns a Boolean value i.e. `true` or `false`.

INCLUDES METHOD OF STRING

The `includes()` method determines whether a string contains a specified substring. It returns `true` if the substring is found, and `false` otherwise.

The `includes()` method takes two parameters:

👉 `searchString`: The substring to search for.

👉 `position` (optional): The index at which to begin the search. If omitted, the search starts at the beginning of the string.

INCLUDES METHOD OF STRING

The **`includes()`** method determines whether a string contains a specified substring. It returns **`true`** if the substring is found, and **`false`** otherwise.

```
const str = "Hello, world!";

console.log(str.includes("world"));   // Output: true
console.log(str.includes("World"));   // Output: false (case-sensitive)
console.log(str.includes("hello"));   // Output: false (case-sensitive)
console.log(str.includes("world", 7)); // Output: true (starts search from
index 7)
console.log(str.includes("!")); // Output: true
console.log(str.includes("")); // Output: true (An empty string is always
included)
```

STARTSWITH METHOD OF STRING

The `startsWith()` method checks if a string begins with a specified substring. It returns `true` if the string starts with the substring, and `false` otherwise.

The `startsWith()` method takes two parameters:

👉 `searchString`: The substring to search for at the beginning of the string.

👉 `position` (optional): The index at which to begin the search. If omitted, the search starts at the beginning of the string.

STARTSWITH METHOD OF STRING

The `startsWith()` method checks if a string begins with a specified substring. It returns `true` if the string starts with the substring, and `false` otherwise.

```
const str = "Hello, world!";

console.log(str.startsWith("Hello"));   // Output: true
console.log(str.startsWith("hello"));   // Output: false (case-sensitive)
console.log(str.startsWith("world"));   // Output: false
console.log(str.startsWith("Hello", 6)); // Output: false (starts searching
from index 6)
console.log(str.startsWith("o", 4)); // Output: true (starts searching from
index 4)
console.log(str.startsWith("")); // Output: true (An empty string is always
at the start)
```

ENDSWITH METHOD OF STRING

The `endsWith()` method checks if a string ends with a specified substring. It returns true if the string ends with the substring, and false otherwise.

The `endsWith()` method takes two parameters:

👉 `searchString`: The substring to search for at the end of the string.

👉 `length` (optional): The length of the string to be searched as if the string is truncated to that length. If omitted, the entire string is searched.

ENDSWITH METHOD OF STRING

The `endsWith()` method checks if a string ends with a specified substring. It returns true if the string ends with the substring, and false otherwise.

```
const str = "Hello, world!";

console.log(str.endsWith("!"));     // Output: true
console.log(str.endsWith("world!"));  // Output: true
console.log(str.endsWith("world"));   // Output: false
console.log(str.endsWith("!", 12));  // Output: false (considers only the
first 12 characters)
console.log(str.endsWith("world", 12)); // Output: true (considers only the
first 12 characters)
console.log(str.endsWith("")); // Output: true (An empty string is always at
the end)
```

PRIMITIVE STRING TYPE VS STRING OBJECT

HOW ARE STRING METHODS CALLED?

The strings are primitive values in JavaScript. Primitive types are immutable and don't have methods in the traditional object-oriented sense. So, how can we call methods like `toUpperCase()`, `slice()`, or `indexOf()` on a string literal or a string variable?

The answer lies in JavaScript's automatic type coercion and the concept of "**wrapper objects**".

HOW ARE STRING METHODS CALLED?

👉 **Automatic Boxing** (or Wrapping): When you try to call a method on a primitive value (like a string, number, or boolean), JavaScript automatically converts that primitive value into a temporary object. This is called "boxing" or "wrapping". For strings, it creates a temporary String object.

👉 **String Object**: This temporary String object is an object wrapper around the primitive string value. It's this String object that has the methods like toUpperCase(), slice(), etc.

👉 **Method Call**: The method is called on this temporary String object.

HOW ARE STRING METHODS CALLED?

👉 **Automatic Unboxing**: After the method call completes, JavaScript automatically converts the String object back to a primitive string value (this is called "unboxing"). The result of the method (which is usually another primitive string) is then returned.

👉 **Garbage Collection**: The temporary String object is then discarded (garbage collected).

In simpler terms: JavaScript "**wraps**" the primitive string in an object just long enough to execute the method, and then "unwraps" it to give you the desired result.

WHAT HAPPENS BEHIND THE SCENES?

```
const str = "hello";
const upperStr = str.toUpperCase();

// What JavaScript does behind the scenes (simplified):

// 1. Boxing:
const tempStringObj = new String("hello"); // Temporary String object

// 2. Method call:
const tempResult = tempStringObj.toUpperCase(); // "HELLO"

// 3. Unboxing:
const upperStr = tempResult.valueOf(); // "HELLO" (primitive string)

// 4. Garbage collection:
// tempStringObj is no longer needed and is garbage collected.

console.log(upperStr); // Output: HELLO
console.log(typeof upperStr); // Output: string (it's a primitive)
```

WHY JAVASCRIPT DO THIS?

This automatic **boxing/unboxing** allows us to treat primitive values as if they were objects, providing them with convenient methods for manipulation. It makes the language easier to use and more consistent.

KEY TAKEAWAY

👉 Even though strings are primitives, JavaScript's automatic type coercion creates temporary wrapper objects when you call methods on them.

👉 This allows you to use object-oriented methods on primitive values without having to explicitly create objects yourself.

👉 The important thing to remember is that the original primitive value is not changed; the methods return new primitive values.

THE PADSTART & PADEND METHODS

PADSTART & PADEND METHODS OF STRING

The `padStart()` and `padEnd()` are string methods in JavaScript introduced in **ES2017** that allow you to pad a string with another string (or spaces) until it reaches a specified length.

The `padStart()` pads at the beginning, and `padEnd()` pads at the end.

They are very useful for formatting strings, especially when you need consistent lengths, like for displaying data in tables or aligning text.

PADSTART METHOD OF STRING

The `padStart()` method pads the beginning of a string with another string (repeated if necessary) until it reaches a given length.

The `padStart()` method takes two arguments:

👉 `targetLength`: The desired length of the resulting string.

👉 `padString` (optional): The string to pad with. If omitted, it defaults to a space.

PADSTART METHOD OF STRING

The `padStart()` method pads the beginning of a string with another string (repeated if necessary) until it reaches a given length.

```
const str = "123";

console.log(str.padStart(5, "0"));   // Output: 00123 (pads with "0" at the
beginning)
console.log(str.padStart(5));       // Output:   123 (pads with spaces at the
beginning)
console.log(str.padStart(8, "*"));   // Output: *****123
console.log(str.padStart(2, "0"));   // Output: 123 (no padding needed,
string is already long enough)
console.log(str.padStart(5, "abc")); // Output: ab123 (pad string is repeated
and truncated if necessary)
console.log(str.padStart(8, "123")); // Output: 12312312
```

PADEND METHOD OF STRING

The `padEnd()` method works similar to `padStart()`, but it pads at the end of a string with another string (repeated if necessary) until it reaches a given length.

The `padEnd()` method takes two arguments:

👉 `targetLength`: The desired length of the resulting string.

👉 `padString` (optional): The string to pad with. If omitted, it defaults to a space.

PADEND METHOD OF STRING

The `padEnd()` method works similar to `padStart()`, but it pads at the end of a string with another string (repeated if necessary) until it reaches a given length.

```
const str = "123";

console.log(str.padEnd(5, "0"));   // Output: 12300 (pads with "0" at the
end)
console.log(str.padEnd(5));       // Output: 123   (pads with spaces at the
end)
console.log(str.padEnd(8, "*"));   // Output: 123*****
console.log(str.padEnd(2, "0"));   // Output: 123 (no padding needed)
console.log(str.padEnd(5, "abc")); // Output: 123ab (pad string is repeated
and truncated if necessary)
console.log(str.padEnd(8, "123")); // Output: 12312312
```

THE REPEAT & CONCAT METHODS

REPEAT METHOD OF STRING

The `repeat()` method constructs and returns a new string which contains the concatenated copies of the string on which it is called, repeated a given number of times.

The `repeat()` method takes a single arguments:

👉 **`count:`** An integer between `0` and `Infinity` (inclusive) indicating the number of times to repeat the string.

REPEAT METHOD OF STRING

The `repeat()` method constructs and returns a new string which contains the concatenated copies of the string on which it is called, repeated a given number of times.

```javascript
const str = "abc";

console.log(str.repeat(0));   // Output: "" (empty string)
console.log(str.repeat(1));   // Output: abc
console.log(str.repeat(2));   // Output: abcabc
console.log(str.repeat(3));   // Output: abcabcabc
console.log(".".repeat(10)); // Output: ..........
```

REPEAT METHOD OF STRING

The `repeat()` method constructs and returns a new string which contains the concatenated copies of the string on which it is called, repeated a given number of times.

```
const str = "abc";

// Error handling:
// RangeError: Invalid count value
console.log(str.repeat(-1));

// RangeError: Invalid count value
console.log(str.repeat(Infinity));
```

REPEAT METHOD OF STRING

The `repeat()` method constructs and returns a new string which contains the concatenated copies of the string on which it is called, repeated a given number of times.

```
const str = "abc";

// Coercion:
console.log(str.repeat(2.5)); // Output: abcabc (count is converted to an
integer)

// Output: abcabcabc (count can be a string that can be converted to a
number)
console.log(str.repeat("3"));
console.log(str.repeat(NaN)); // Output: "" (NaN is treated as 0)
```

CONCAT METHOD OF STRING

The `concat()` method combines one or more strings to create a new string. It's functionally similar to the `+` operator for string concatenation but can be used to concatenate multiple strings at once.

CONCAT METHOD OF STRING

The **concat()** method combines one or more strings to create a new string. It's functionally similar to the **+** operator for string concatenation but can be used to concatenate multiple strings at once.

```
const str1 = "Hello";
const str2 = " ";
const str3 = "world!";

console.log(str1.concat(str2, str3));    // Output: Hello world!
console.log(str1.concat(str2, str3, " How are you?")); // Output: Hello
world! How are you?
console.log("".concat("This", " ", "is", " ", "a", " ", "string.")); //
Output: This is a string.
```

THE GROUPBY METHOD IN JAVASCRIPT

THE GROUPBY METHOD

The ES2024 update introduces a new static method, `Object.groupBy()`, that provides a standardized way to group elements of an iterable (like an array) into an object based on a grouping criterion.

The `Object.groupBy()` takes two arguments:

👉 The **iterable**: The array or other iterable you want to group.

👉 A **callback function**: This function is called for each element in the iterable. It should return the key by which you want to group the element.

THE GROUPBY METHOD

The ES2024 update introduces a new static method, `Object.groupBy()`, that provides a standardized way to group elements of an iterable (like an array) into an object based on a grouping criterion.

```
const numbers = [2, 4, 3, 7, 8, 16, 31, 28];

const group = Object.groupBy(numbers, number => {
    return number % 2 === 0 ? "even" : "odd";
});

console.log(group); // {“even”: [2, 4, 8, 16, 26], “odd”: [3, 7, 31]}
```

THE GROUPBY METHOD

The ES2024 update introduces a new static method, `Object.groupBy()`, that provides a standardized way to group elements of an iterable (like an array) into an object based on a grouping criterion.

```javascript
const people = [
  { name: 'Alice', age: 25, city: 'New York' },
  { name: 'Bob', age: 30, city: 'London' },
  { name: 'Charlie', age: 25, city: 'New York' },
  { name: 'David', age: 35, city: 'Paris' },
  { name: 'Eve', age: 30, city: 'London' },
];

const peopleByCity = Object.groupBy(people, person => person.city);

console.log(peopleByCity);
```

ARRAY FILL & FROM METHODS

THE ARRAY FILL METHOD

The `fill()` method modifies an existing array by filling all or a portion of its elements with a static value.

The `Array.fill()` method takes three arguments:

- 👉 `value`: The value to fill the array with.
- 👉 `start` (optional): The index to start filling from (inclusive). Defaults to 0.
- 👉 `end` (optional): The index to stop filling at (exclusive). Defaults to array.length.

THE ARRAY FILL METHOD

The `fill()` method modifies an existing array by filling all or a portion of its elements with a static value.

Points to Remember:

👉 **Modifies the original array**: `fill()` changes the array directly. It does not create a new array.
👉 **Sparse arrays**: `fill()` can also be used with sparse arrays. It will fill the empty slots as well.

THE ARRAY FILL METHOD

The `fill()` method modifies an existing array by filling all or a portion of its elements with a static value.

```
const arr = [1, 2, 3, 4];
arr.fill(0); // Fills the entire array with 0
console.log(arr); // Output: [0, 0, 0, 0]

arr.fill(0, 1, 3); // Fill from index 1 (inclusive) to 3 (exclusive)
console.log(arr); // Output: [1, 0, 0, 4]

arr.fill(0, 2); // Fill from index 2 to the end
console.log(arr); // Output: [1, 2, 0, 0]

arr.fill(0, -2); // Fill from the last second element
console.log(arr); // Output: [1, 2, 0, 0]

arr2.fill(0, -3, -1); // Fills from index -3 (inclusive) to -1 (exclusive)
console.log(arr2); // Output: [1, 0, 0, 4]
```

THE ARRAY FROM METHOD

The `from()` method creates a new array from an iterable object (like a `string`, `Map`, `Set`, or an array-like object) or any object with a length property.

The `Array.from()` method takes three arguments:

👉 `arrayLike`: The iterable or array-like object to convert to an array.

👉 `mapFn` (optional): A function to call on each element of the array-like object before adding it to the new array. This is like the `map()` method.

👉 `thisArg` (optional): Value to use as this when executing mapFn.

THE ARRAY FROM METHOD

The `from()` method creates a new array from an iterable object (like a `string`, `Map`, `Set`, or an array-like object) or any object with a length property.

```javascript
const arr = Array.from('hello');
console.log(arr); // Output: ['h', 'e', 'l', 'l', 'o']

const mySet = new Set([1, 2, 3]);
const arr = Array.from(mySet);
console.log(arr); // Output: [1, 2, 3]

const arr = Array.from([1, 2, 3], x => x * 2); // Map each element
console.log(arr); // Output: [2, 4, 6]
```