

1. every( ) .....	2
2. some( ) .....	3
3. fill( ) .....	3
Array methods .....	4
A. Add/remove items .....	4
1. arr.push .....	4
2. arr.unshift .....	4
3. arr.pop( .....	4
4. arr.shift.....	4
5. splice .....	4
Syntax.....	4
6. slice .....	5
Syntax.....	5
7. String concat .....	5
Syntax.....	5
B. Iterate: forEach .....	6
★ Comparing forEach() with a for Loop.....	9
★ How forEach() loop is different from for loop method? .....	11
C. Searching in array .....	12
1. indexOf/lastIndexOf and includes .....	12
2. find and findIndex/findLastIndex.....	15
The find method .....	15
The findIndex method .....	17
3. filter .....	18
D. Transform an array .....	19
1. map .....	19
2. sort(fn) .....	20
3. reverse .....	20
Numeric sort .....	20
4. split and join.....	21
5. reduce/reduceRight\ .....	22
reduce() Syntax .....	22
reduceRight() .....	26

reduceRight() Syntax.....	27
F. Summary .....	28
G. Arrow Function .....	30
Arrow Function Syntax.....	30
Example 1: Arrow Function with No Argument.....	31
Example 2: Arrow Function with One Argument .....	31
Example 3: Arrow Function as an Expression .....	31
Example 4: Multiline Arrow Functions .....	31
this with Arrow Function .....	32
Arguments Binding .....	34
Arrow Function with Promises and Callbacks .....	34
Things You Should Avoid With Arrow Functions .....	35
H. How to use different array methods with arrow functions ? .....	35
I. JSON in JavaScript .....	37
J. localStorage with example.....	39
Example: Perform CRUD operation using localStorage methods.....	40
L. array Vs object in javascript .....	44
★ Accessing Nested Arrays .....	45
★ how to Nested array and object with each other, Give example.....	46
Spread operator (3 dot) .....	48

## JavaScript Array Methods

### 1. every( )

This method checks every element in the array that passes the condition, returning true or false as appropriate.

```
1  const arr = [1, 2, 3, 4, 5, 6, 7];
2
3  // all elements are greater than 5
4  const greaterFive = arr.every(num => num > 5);
5  console.log(greaterFive); // false
6
7  // all elements are less than 9
8  const lessnine = arr.every(num => num < 9);
9  console.log(lessnine); // true
```

## 2. some( )

This method checks if at least one element in the array that passes the condition, returning true or false as appropriate.

```
1  const arr = [1, 2, 3, 4, 5, 6, 7];
2
3  // at least one element is greater than 5?
4  const greaterNum = arr.some(num => num > 5);
5  console.log(greaterNum); // true
6
7  // at least one element is less than or equal to 0?
8  const lessNum = arr.some(num => num <= 0);
9  console.log(lessNum); // false
```

## 3. fill( )

This method fills the elements in an array with a static value and returns the modified array.

```
1  const arr = new Array(3);
2  console.log(arr); // [empty, empty, empty]
3  console.log(arr.fill(10)); // [10, 10, 10]
```

# Array methods

Arrays provide a lot of methods. To make things easier, in this chapter they are split into groups.

Syntax for creating array:

```
const array_name = [item1, item2, ...];
```

## A. Add/remove items

1. `arr.push(...items)` – adds items to the end,
2. `arr.unshift(item1, item2, ..., itemX)` – The `unshift()` method adds new elements to **the beginning** of an array. The `unshift()` method overwrites the original array.
3. `arr.pop()` – extracts an item from the end,
4. `arr.shift()` – Shift (remove) the first element of the array. The `shift()` method returns the shifted element i.e last element.

Here are a few others.

## 5. splice

### *Syntax*

```
array.splice(index, howmany, item1, ....., itemX)
```

1. The `splice()` method adds and/or removes array elements.
2. The `splice()` method overwrites the original array.

## Examples

- At position 2, add 2 elements:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi"); // Banana,Orange,Lemon,Kiwi,Apple,Mango
```

- At position 2, remove 2 items:

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi"];
fruits.splice(2, 2); // Banana,Orange,Kiwi
```

- **Negative indexes allowed**

Here and in other array methods, negative indexes are allowed. They specify the position from the end of the array, like here:

```
let arr = [1, 2, 5];

// from index -1 (one step from the end)
// delete 0 elements,
// then insert 3 and 4
arr.splice(-1, 0, 3, 4);
alert( arr ); // 1,2,3,4,5
```

## 6. slice

- `slice()` method returns a new array with specified start to end elements.
- The `slice()` method does not change the original array.

### Syntax

`array.slice(start, end)`

## Examples

- Select elements:

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1, 3); // Orange,Lemon
```

- Select elements using negative values:

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const myBest = fruits.slice(-3, -1); // Lemon,Apple
```

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

## 7. String concat

- The `concat()` method joins two or more strings.
- The `concat()` method does not change the existing strings.
- The `concat()` method returns a new string.

### Syntax

`string.concat(string1, string2, ..., stringX)`

## Examples

- Join two strings:

```
let text1 = "sea";
let text2 = "food";
let result = text1.concat(text2); // seafood
```

- Join two strings:

```
let text1 = "Hello";
let text2 = "world!";
let result = text1.concat(" ", text2); // Hello world!
```

## B. Iterate: forEach

<https://www.freecodecamp.org/news/javascript-foreach-js-array-for-each-example/>

- The `forEach()` method calls a function for each element in an array.
- This function is referred to as a callback function.
- The `forEach()` method is not executed for empty elements.

syntax:

```
array.forEach(callbackFunction);
or
arr.forEach(function(item, index, array) {
  // ... do something with item
});
```

- The callback function can accept up to three different arguments, though not all of them are required.

Example

```
const staffsDetails = [
  { name: "Jam Josh", age: 44, salary: 4000, currency: "USD" },
  { name: "Justina Kap", age: 34, salary: 3000, currency: "USD" },
  { name: "Chris Colt", age: 37, salary: 3700, currency: "USD" },
  { name: "Jane Doe", age: 24, salary: 4200, currency: "USD" }
];

staffsDetails.forEach((staffDetail) => {
  let sentence = `I am ${staffDetail.name} a staff of Royal Suites.`;
  console.log(sentence);
});
```

## Output:

```
"I am Jam Josh a staff of Royal Suites."  
"I am Justina Kap a staff of Royal Suites."  
"I am Chris Colt a staff of Royal Suites."  
"I am Jane Doe a staff of Royal Suites."
```

```
staffsDetails.forEach((staffDetail, index) => {  
  let sentence = `index ${index} : I am ${staffDetail.name} a staff of  
  Royal Suites.`;  
  console.log(sentence);  
});
```

## Output:

```
"index 0 : I am Jam Josh a staff of Royal Suites."  
"index 1 : I am Justina Kap a staff of Royal Suites."  
"index 2 : I am Chris Colt a staff of Royal Suites."  
"index 3 : I am Jane Doe a staff of Royal Suites."
```

```
staffsDetails.forEach((staffDetail, index, array) => {  
  console.log(array);  
});
```

Example 2:

```
let scores = [12, 55, 70];  
  
scores.forEach((score, index, array) => {  
  console.log(array);  
});
```

Output:

```
[12,55,70]  
[12,55,70]  
[12,55,70]
```

## ★ How to Add All Values in An Array of Numbers with forEach()

```
const scores = [12, 55, 70, 47];  
let total = 0;  
scores.forEach((score) => {
```

```
    total += score;  
  });
```

```
console.log(total);
```

Recall that earlier on, we were making use of an array of staff details. Now let's try adding all the staff member's salaries together to see how it works with objects:

```
let totalSalary = 0;  
staffsDetails.forEach(({salary}) => {  
    totalSalary += salary;  
});  
  
console.log(totalSalary + " USD"); // "14900 USD"
```

## ★ How to Use Conditionals in a `forEach()` Callback Function

When looping through arrays, we may want to check for specific conditions, as is commonly done with the `for` loop method. We can pass these conditions into our callback function or any other operation we want to run on each array item.

For example, if we only want to show the names of people whose salaries are greater than or equal to 4000 from the array of staff details we declared earlier, we can do the following:

```
staffsDetails.forEach(({name, salary}) => {  
    if(salary >= 4000){  
        console.log(name);  
    }  
});
```

Output:

```
"Jam Josh"  
"Jane Doe"
```



## ★ Comparing `forEach()` with a `for` Loop

The `for` loop is very similar to the `forEach` method, but each possess some features that are unique to them such as:

### Break out and continue in a Loop

- When looping through an array, we may want to either break out or continue the loop when a certain condition is met (meaning we skip). This is possible with the `break` and `continue` instruction, but it does not work with the `forEach()` method, as shown below:

```
const scores = [12, 55, 70, 47];

scores.forEach((score) => {
  console.log(score);

  if (score === 70)
    break;
});
```

- This will throw a syntax error of `Illegal break statement`. This applies also to the `continue` instruction which would also throw an `Illegal continue statement: no surrounding iteration statement`.

```
const scores = [12, 55, 70, 47];

scores.forEach((score) => {
  if (score === 70)
    continue;

  console.log(score);
});
```

But fortunately, this works with the `for` loop method perfectly:

```
const scores = [12, 55, 70, 47];
for (i = 0; i < scores.length; i++) {
  console.log(scores[i]);
}
```

```
    if (scores[i] === 70)
        break;
}
```

Output:

```
12
55
70
```

And the same with the continue instruction:

```
const scores = [12, 55, 70, 47];

for (i = 0; i < scores.length; i++) {
    if (scores[i] === 70)
        continue;

    console.log(scores[i]);
}
```

Output:

```
12
55
47
```

## Array with Missing elements

- Another important comparison to make is in a situation whereby the array we are iterating over has some missing values/array items as seen below:

```
const studentsScores = [70, , 12, 55, , 70, 47];
```

This could be due to a developer error or something else, but these two methods take two different approaches to looping through these types of arrays. The for loop returns undefined where there are missing values, whereas the forEach() method skips them.

### For Loop

```
const studentsScores = [70, , 12, 55, , 70, 47];

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

Output:

70  
undefined  
12  
55  
undefined  
70  
47

## forEach()

```
const studentsScores = [70, , 12, 55, , 70, 47];  
  
studentsScores.forEach((studentScore) => {  
  console.log(studentScore);  
});
```

Output:

70  
12  
55  
70  
47

**Note:** Async/Await does not work with the `forEach()` array method but works with the `for` loop method.

## Conclusion

- In this article, we learned how to use the `forEach()` array method, which allows us to loop through an array of any type of item. It also allows us to write cleaner, more readable code than the `for` loop.

### ★ How `forEach()` loop is different from `for` loop method?

**For Loop:** The JavaScript `for` loop is used to iterate through the array or the elements for a specified number of times. If a certain amount of iteration is known, it should be used.

**forEach loop:** The `forEach()` method is also used to loop through arrays, but it uses a function differently than the classic “for loop”. It passes a callback function for each element of an array together with the below parameters:

8. **Current Value (required):** The value of the current array element

9. **Index (optional):** The index number of the current element

10. **Array (optional):** The array object the current element belongs to

We need a callback function to loop through an array by using the `forEach` method.

## C. Searching in array

Now let's cover methods that search in an array.

### 1. `indexOf/lastIndexOf` and `includes`

The methods `arr.indexOf` and `arr.includes` have the similar syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

11. `arr.indexOf(item, from)` – to find out whether the element is present in the array or not. But it doesn't return a Boolean. **It returns the first index of the element found in the array, or it will return -1 (which represents that the element is not found).**

12. `arr.includes(item, from)` – This method is used to find out whether the element is included in the array or not. It returns a Boolean value — `true` or `false`.

Usually these methods are used with only one argument: the `item` to search. By default, the search is from the beginning.

Let's perform a certain operation on both methods.

For instance:

```
let arr = [1, 0, false];
```

```
alert( arr.indexOf(0)); // 1
```

```
alert( arr.indexOf(false)); // 2
```

```
alert( arr.indexOf(null)); // -1
```

```
alert( arr.includes(1)); // true
```

## JavaScript

Please note that `indexOf` uses the strict equality `===` for comparison. So, if we look for `false`, it finds exactly `false` and not the zero.

If we want to check if item exists in the array, and don't need the index, then `arr.includes` is preferred.

The method [arr.lastIndexOf](#) is the same as `indexOf`, but looks for from right to left.

```
let fruits = ['Apple', 'Orange', 'Apple']
```

```
alert( fruits.indexOf('Apple') ); // 0 (first Apple)
alert( fruits.lastIndexOf('Apple') ); // 2 (last Apple)
```

### **The includes method handles NaN correctly**

A minor, but noteworthy feature of `includes` is that it correctly handles `NaN`, unlike `indexOf`:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (wrong, should be 0)
alert( arr.includes(NaN) ); // true (correct)
```

That's because `includes` was added to JavaScript much later and uses the more up to date comparison algorithm internally.

### **Another Examples for “includes” and “indexOf”**

<https://medium.com/@kanzariyamihir/includes-and-indexof-51485dc2b871>

## **Includes()**

```
const array = [5,11, 6, 2, 13, 7, 9, 25, 88]
if (array.includes(2)) { console.log(`array contain 2`) // array contain 2 }
```

## **indexOf()**

```
const array = [5,11, 6, 2, 13, 7, 9, 25, 88]
if (array.indexOf(2) >= -1) { console.log(`array contain 2`) // array contain 2 }
```

Let's check for `NaN`.

## Includes()

```
const array = [5,11, 6, 2, NaN, 7, 9, 25, 88]if
(array.includes(NaN)) {    console.log(`array contain NaN`) //
array contain NaN}
```

## indexOf()

```
const array = [5,11, 6, 2, NaN, 7, 9, 25, 88]if
(array.indexOf(NaN) >= -1) {    console.log(`array contain NaN`)
// array contain NaN}
```

Let's check for undefined.

## Includes()

```
const array = [5,11, 6, 2, undefined, 7, 9, 25, 88]If
(array.includes(undefined)) {    console.log("elements from
array are undefined")
    // elements from array are undefined}
```

## indexOf()

```
const array = [5,11, 6, 2, undefined, 7, 9, 25, 88]if
(!array.indexOf(undefined) == -1) {    console.log(`elements
from array are undefined`)} else {    console.log(`Can't find
undefined.`) // Can't find undefined.}
```

With negative/positive/zero.

```
const a = [-0].includes(0) console.log(a) // true
const b = [-0].indexOf(0) >= -1 console.log(b) //true
const c = [-1].includes(1) console.log(c) // false
const d = [-1].indexOf(1) >= -1 console.log(d) // true
```

## Summary:

Includes() Method works with NaN, Zero ,undefined .

indexOf() method works with NaN, Zero but not with undefined.

If you type compare -0 with +0 that is true.

```
console.log(-0 === +0) /// true
```

## **2. find and findIndex/findLastIndex**

Imagine we have an array of objects. How do we find an object with the specific condition? Then we use find() method.

### The find method

- The method `find()` is used to return the first array element that passes a certain test function.
- The find method takes a function as its parameter and the function itself takes three arguments: the element, the index, and the array itself.

Here the [arr.find\(fn\)](#) method comes in handy.

The syntax is:

```
let result = arr.find(function(item, index, array) {  
  // if true is returned, item is returned and iteration is stopped  
  // for falsy scenario returns undefined  
});
```

The function is called for elements of the array, one after another:

13. `item` is the element.
14. `index` is its index.
15. `array` is the array itself.

If it returns true, the search is stopped, the `item` is returned. If nothing found, `undefined` is returned.

## JavaScript

For example, we have an array of users, each with the fields `id` and `name`. Let's find the one with `id == 1`:

```
let users = [  
  {id: 1, name: "John"},  
  {id: 2, name: "Pete"},  
  {id: 3, name: "Mary"}  
];  
  
let user = users.find(item => item.id == 1);
```

```
alert(user.name); // John
```

In real life arrays of objects is a common thing, so the `find` method is very useful.

Note that in the example we provide to `find` the function `item => item.id == 1` with one argument. That's typical, other arguments of this function are rarely used.

The [arr.findIndex](#) method has the same syntax but returns the index where the element was found instead of the element itself. The value of `-1` is returned if nothing is found.

The [arr.findLastIndex](#) method is like `findIndex`, but searches from right to left, similar to `lastIndexOf`.

Here's an example:

```
let users = [  
  {id: 1, name: "John"},  
  {id: 2, name: "Pete"},  
  {id: 3, name: "Mary"},  
  {id: 4, name: "John"}  
];  
  
// Find the index of the first John  
alert(users.findIndex(user => user.name == 'John')); // 0  
  
// Find the index of the last John  
alert(users.findLastIndex(user => user.name == 'John')); // 3
```

**Another reference of understanding for “find” and “findIndex”:**

<https://javascript.plainenglish.io/difference-between-find-and-findindex-in-javascript-63204178bbc1#:~:text=passes%20the%20test.-,The%20Differences,method%20returns%20the%20element%20index.>



## Let's have a look at a practical example:

```
let numbers = [2, 15, 18, 25]; //using ES5
numbers.find(function(element) {
  return element >= 18;
}); //output: 18
//The same example using ES6
numbers.find(element => element >= 18); //output: 18
```

As you can see above, The method `find()` tries to find in the array the first element greater than or equal to 18, then it returns it.

Also, keep in mind that the find method does not change the original array `numbers`.

Here is an example:

```
console.log(numbers); //output: [2, 15, 18, 25]
```

The method `find()` returns `undefined` if none of the elements in the array passes the test. Also, for simplicity, we only used one parameter in the callback function that we passed to the find method as a parameter.

```
2 | const found = arr.find(element => element > 5);
3 | console.log(found); // 6
```

## The `findIndex` method

The method `findIndex` is used to return the index of the first array element that passes a certain test function.

The method `findIndex` also takes a callback function which can take three parameters(`element`, `index`, and `array`).

Here is an example:

```
let numbers = [2, 10, 18, 25]; //using ES5
numbers.findIndex(function(element) {
  return element > 2;
}); //output: 1
```

## JavaScript

```
//The same example using ES6  
numbers.findIndex(element => element > 2); //output: 1
```

As you can see above, the method `findIndex()` tries to find the index of the first element greater than 2, then it returns it. The first element that is greater than 2 in the array is 10 and its index is 1. That's why we get 1 in the output.

The method `findIndex()` does not change the array too. Here is an example:

```
console.log(numbers); //output: [2, 10, 18, 25]
```

The method `findIndex` also returns `undefined` if none of the elements in the array passes the test.

```
2   const indexFinder = arr.findIndex(element => element === "Mandeep");  
3   console.log(indexFinder); // 1
```

## The Differences

The method `find()` is very similar to `findIndex()`. The only difference is that the `find` method returns the element value, but the `findIndex` method returns the element index.

### 3. [filter](#)

- The `find` method looks for a single (first) element that makes the function return `true`.
- If there may be many, we can use [arr.filter\(fn\)](#).
- The syntax is similar to `find`, but `filter` returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {  
  // if true item is pushed to results and the iteration continues  
  // returns empty array if nothing found  
});
```

For instance:

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

## D. Transform an array

Let's move on to methods that transform and reorder an array.

### 1. map

- [arr.map](#) method calls the function for each element of the array and returns the array of results.
- `map()` does not execute the function for empty elements.
- `map()` does not change the original array.

The syntax is:

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});
```

For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

Example 2:

```
<script>
const persons = [
  {firstname : "Malcom", lastname: "Reynolds"},
  {firstname : "Kaylee", lastname: "Frye"},
  {firstname : "Jayne", lastname: "Cobb"}
];

document.getElementById("demo").innerHTML = persons.map(getFullName);
```

## JavaScript

```
function getFullName(item) {  
    return [item.firstname,item.lastname].join(" ");  
}  
</script>
```

Output:

Malcom Reynolds,Kaylee Frye,Jayne Cobb

Example 3:

```
const array1 = [1, 4, 9, 16];
```

```
// Pass a function to map  
const map1 = array1.map(x => x * 2);
```

```
console.log(map1);  
// Expected output: Array [2, 8, 18, 32]
```

Example 4:

## 2. sort(fn)

[https://www.w3schools.com/js/js\\_array\\_sort.asp](https://www.w3schools.com/js/js_array_sort.asp)

- The `sort()` method sorts an array alphabetically:

### Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort(); // Apple,Banana,Mango,Orange
```

## 3. reverse

- The `reverse()` method reverses the elements in an array.
- You can use it to sort an array in descending order:

### Example

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.sort(); // Banana,Orange,Apple,Mango  
fruits.reverse(); // Orange,Mango,Banana,Apple
```

## Numeric sort

By default, the `sort()` function sorts values as **strings**.

This works well for strings ("Apple" comes before "Banana").

However, if numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1".

Because of this, the `sort()` method will produce incorrect result when sorting numbers.

You can fix this by providing a **compare function**:

### Example

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b}); // 1,5,10,25,40,100
```

## 4. split and join

[https://www.w3schools.com/jsref/jsref\\_split.asp](https://www.w3schools.com/jsref/jsref_split.asp)

- The `split()` method splits a string into an array of substrings.
- The `split()` method returns the new array.
- The `split()` method does not change the original string.

If (" ") is used as separator, the string is split between words.

### Examples

- Split the words:

```
let text = "How are you doing today?";
const myArray = text.split(" "); // How,are,you,doing,today?
```

- Split the words, and return the second word:

```
let text = "How are you doing today?";
const myArray = text.split(" ");
let word = myArray[1]; // are
```

- Split the characters, including spaces:

```
const myArray = text.split(""); // H,o,w, ,a,r,e, ,y,o,u, ,d,o,i,n,g, ,t,o,d,a,y,?
```

- Use the limit parameter:

```
const myArray = text.split(" ", 3); // How,are,you
```

## Join

- The `join()` method returns an array as a string.
- The `join()` method does not change the original array.
- Any separator can be specified. The default is comma (,).

## Examples

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = fruits.join(); // Banana,Orange,Apple,Mango
```

- Another separator:

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let text = fruits.join(" and "); // Banana and Orange and Apple and Mango
```

## 5. [reduce/reduceRight](https://www.programiz.com/javascript/library/array/reduce)

<https://www.programiz.com/javascript/library/array/reduce>

- The `reduce()` method executes a reducer function on each element of the array and returns a single output value.

## Example

```
const message = ["JavaScript ", "is ", "fun."];

// function to join each string elements
function joinStrings(accumulator, currentValue) {
  return accumulator + currentValue;
}

// reduce join each element of the string
let joinedString = message.reduce(joinStrings);
console.log(joinedString);

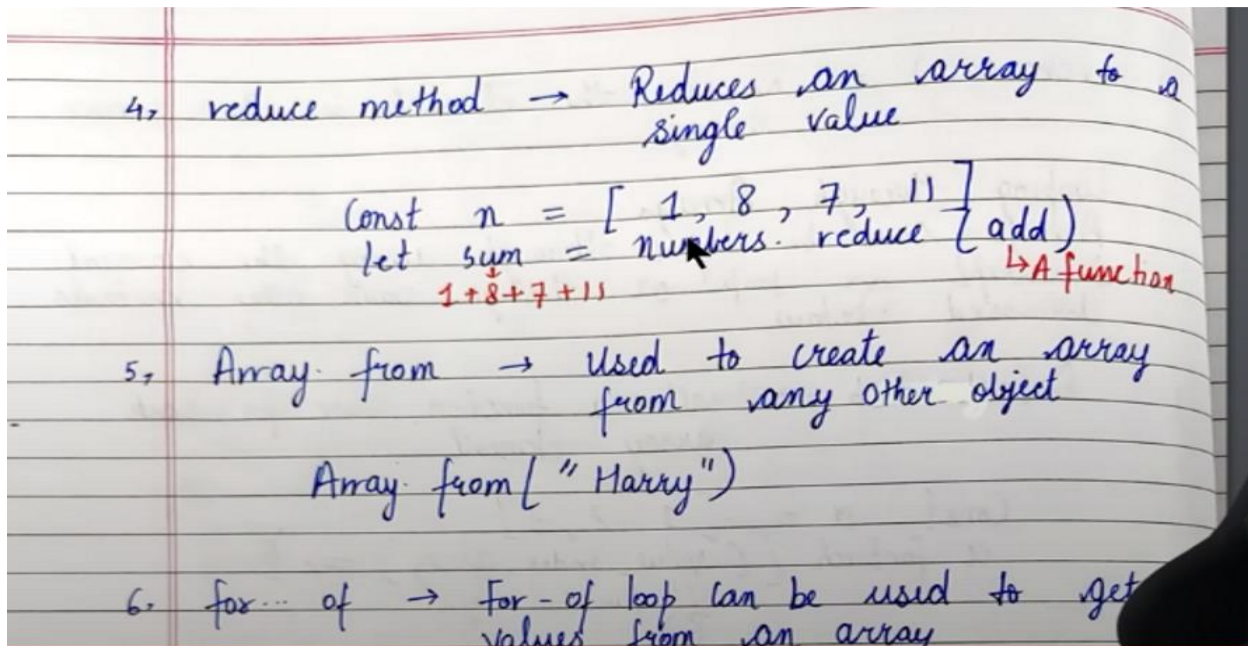
// Output: JavaScript is fun.
```

## *reduce() Syntax*

The syntax of the `reduce()` method is:

```
arr.reduce(callback(accumulator, currentValue), initialValue)
```

Here, arr is an array.



## reduce() Parameters

The `reduce()` method takes in:

- callback - The function to execute on each array element (except the first element if no initialValue is provided). It takes in
  - accumulator - It accumulates the callback's return values.
  - currentValue - The current element being passed from the array.
- 16. initialValue (optional) - A value that will be passed to `callback()` on first call. If not provided, the first element acts as the accumulator on the first call and `callback()` won't execute on it.

**Note:** Calling `reduce()` on an empty array without initialValue will throw `TypeError`.

## reduce() Return Value

17. Returns the single value resulting after reducing the array.

**Notes:**

18. `reduce()` executes the given function for each value from left to right.
19. `reduce()` does not change the original array.
20. It is almost always safer to provide `initialValue`.

```
1  const arr = [1, 2, 3, 4, 5, 6];
2  const reduced = arr.reduce((total, current) => total + current);
3  console.log(reduced); // 21
```

### Example 1: Sum of All Values of Array

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

function sum_reducer(accumulator, currentValue) {
  return accumulator + currentValue;
}

let sum = numbers.reduce(sum_reducer);

console.log(sum); // 21
// using arrow function

let summation = numbers.reduce(
  (accumulator, currentValue) => accumulator + currentValue
);

console.log(summation); // 21
```

### Output

21

21

### Example 2: Subtracting Numbers in Array

```
const numbers = [1800, 50, 300, 20, 100];
```



## JavaScript

```
// subtract all numbers from first number
// since 1st element is called as accumulator rather than currentValue
// 1800 - 50 - 300 - 20 - 100
let difference = numbers.reduce(
  (accumulator, currentValue) => accumulator - currentValue
);
console.log(difference); // 1330
```

```
const expenses = [1800, 2000, 3000, 5000, 500];
const salary = 15000;
```

```
// function that subtracts all array elements from given number
// 15000 - 1800 - 2000 - 3000 - 5000 - 500
let remaining = expenses.reduce(
  (accumulator, currentValue) => accumulator - currentValue,
  salary
);
console.log(remaining); // 2700
```

### Output

```
1330
2700
```

This example clearly explains the difference between passing an initialValue and not passing an initialValue.

### Example 3: Remove Duplicate Items from Array

```
let ageGroup = [18, 21, 1, 1, 51, 18, 21, 5, 18, 7, 10];
let uniqueAgeGroup = ageGroup.reduce(function (accumulator, currentValue) {
  if (accumulator.indexOf(currentValue) === -1) {
    accumulator.push(currentValue);
  }
  return accumulator;
}, []);
```

```
console.log(uniqueAgeGroup); // [ 18, 21, 1, 51, 5, 7, 10 ]
```

### Output

```
[
  18, 21, 1, 51,
  5, 7, 10
]
```

## Example 4: Grouping Objects by a property

```
let people = [
  { name: "John", age: 21 },
  { name: "Oliver", age: 55 },
  { name: "Michael", age: 55 },
  { name: "Dwight", age: 19 },
  { name: "Oscar", age: 21 },
  { name: "Kevin", age: 55 },
];

function groupBy(objectArray, property) {
  return objectArray.reduce(function (accumulator, currentObject) {
    let key = currentObject[property];
    if (!accumulator[key]) {
      accumulator[key] = [];
    }
    accumulator[key].push(currentObject);
    return accumulator;
  }, {});
}

let groupedPeople = groupBy(people, "age");
console.log(groupedPeople);
```

## Output

```
{
  '19': [ { name: 'Dwight', age: 19 } ],
  '21': [ { name: 'John', age: 21 }, { name: 'Oscar', age: 21 } ],
  '55': [
    { name: 'Oliver', age: 55 },
    { name: 'Michael', age: 55 },
    { name: 'Kevin', age: 55 }
  ]
}
```

## reduceRight()

- The `reduceRight()` method reduces the array to a single value by executing a callback function on two values of the array (from right to left).

## Example

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

// function that adds last two values of the numbers array
function sum_reducer(accumulator, currentValue) {
  return accumulator + currentValue;
}

// returns a single value after reducing the numbers array
let sum = numbers.reduceRight(sum_reducer);

console.log(sum);

// Output: 10
```

### *reduceRight() Syntax*

The syntax of the `reduceRight()` method is:

`arr.reduceRight(callback(accumulator, currentValue), initialValue)`

Here, arr is an array.

## **E. Array.isArray**

Arrays do not form a separate language type. They are based on objects.

So `typeof` does not help to distinguish a plain object from an array:

```
alert(typeof {}); // object
alert(typeof []); // object (same)
```

...But arrays are used so often that there's a special method for that: [Array.isArray\(value\)](#).

It returns `true` if the value is an array, and `false` otherwise.

```
alert(Array.isArray({})); // false
```

```
alert(Array.isArray([])); // true
```

## **F. Summary**

A cheat sheet of array methods:

### **To add/remove elements:**

- 21. `push(...items)` – adds items to the end,
- 22. `pop()` – extracts/remove an item from the end,
- 23. `shift()` – extracts/remove an item from the beginning,
- 24. `unshift(...items)` – adds items to the beginning.
- 25. `splice(pos, deleteCount, ...items)` – at index `pos` deletes `deleteCount` elements and inserts `items`.
- 26. `slice(start, end)` – creates a new array, copies elements from index `start` till `end` (not inclusive) into it.
- 27. `concat(...items)` – returns a new array: copies all members of the current one and adds `items` to it. If any of `items` is an array, then its elements are taken.

### **To search among elements:**

- 28. `indexOf/lastIndexOf(item, pos)` – look for `item` starting from position `pos`, return the index or `-1` if not found.
- 29. `includes(value)` – returns `true` if the array has `value`, otherwise `false`.
- 30. `find/filter(func)` – filter elements through the function, return first/all values that make it return `true`.
- 31. `findIndex` is like `find`, but returns the index instead of a value.

### **To iterate over elements:**

- 32. `forEach(func)` – calls `func` for every element, does not return anything.

### **To transform the array:**

- 33. `map(func)` – creates a new array from results of calling `func` for every element.
- 34. `sort(func)` – sorts the array in-place, then returns it.
- 35. `reverse()` – reverses the array in-place, then returns it.
- 36. `split/join` – convert a string to array and back.
- 37. `reduce/reduceRight(func, initial)` – calculate a single value over the array by calling `func` for each element and passing an intermediate result between the calls.

## Additionally:

38. `Array.isArray(value)` checks value for being an array, if so returns true, otherwise false.

Please note that methods `sort`, `reverse` and `splice` modify the array itself.

These methods are the most used ones, they cover 99% of use cases. But there are few others:

39. [`arr.some\(fn\)`](#)/[`arr.every\(fn\)`](#) check the array.

The function `fn` is called on each element of the array similar to `map`. If any/all results are true, returns true, otherwise false.

These methods behave sort of like `||` and `&&` operators: if `fn` returns a truthy value, `arr.some()` immediately returns true and stops iterating over the rest of items; if `fn` returns a falsy value, `arr.every()` immediately returns false and stops iterating over the rest of items as well.

We can use `every` to compare arrays:

```
function arraysEqual(arr1, arr2) {  
  return arr1.length === arr2.length && arr1.every((value, index) => value  
    === arr2[index]);  
}
```

```
alert( arraysEqual([1, 2], [1, 2])); // true
```

40. [`arr.fill\(value, start, end\)`](#) – fills the array with repeating value from index `start` to `end`.

41. [`arr.copyWithin\(target, start, end\)`](#) – copies its elements from position `start` till position `end` into *itself*, at position `target` (overwrites existing).

42. [`arr.flat\(depth\)`](#)/[`arr.flatMap\(fn\)`](#) create a new flat array from a multidimensional array.

## G. Arrow Function

Arrow function is one of the features introduced in the ES6 version of JavaScript. It allows you to create functions in a cleaner way compared to regular functions. For example,

This function

```
// function expression
let x = function(x, y) {
    return x * y;
}
```

can be written as

```
// using arrow functions
let x = (x, y) => x * y;
```

using an arrow function.

## Arrow Function Syntax

The syntax of the arrow function is:

```
let myFunction = (arg1, arg2, ...argN) => {
    statement(s)
}
```

Here,

- 43. myFunction is the name of the function
- 44. arg1, arg2, ...argN are the function arguments
- 45. statement(s) is the function body

If the body has single statement or expression, you can write arrow function as:

```
let myFunction = (arg1, arg2, ...argN) => expression
```

## Example 1: Arrow Function with No Argument

If a function doesn't take any argument, then you should use empty parentheses. For example,

```
let greet = () => console.log('Hello');  
greet(); // Hello
```

## Example 2: Arrow Function with One Argument

If a function has only one argument, you can omit the parentheses. For example,

```
let greet = x => console.log(x);  
greet('Hello'); // Hello
```

## Example 3: Arrow Function as an Expression

You can also dynamically create a function and use it as an expression. For example,

```
let age = 5;  
  
let welcome = (age < 18) ?  
  () => console.log('Baby') :  
  () => console.log('Adult');  
  
welcome(); // Baby
```

## Example 4: Multiline Arrow Functions

If a function body has multiple statements, you need to put them inside curly brackets {}. For example,

```
let sum = (a, b) => {  
  let result = a + b;
```

## JavaScript

```
        return result;
    }

    let result1 = sum(5,7);
    console.log(result1); // 12
```

## this with Arrow Function

Inside a regular function, [this keyword](#) refers to the function where it is called.

However, `this` is not associated with arrow functions. Arrow function does not have its own `this`. So whenever you call `this`, it refers to its parent scope. For example,

## Inside a regular function

```
function Person() {
    this.name = 'Jack',
    this.age = 25,
    this.sayName = function () {

        // this is accessible
        console.log(this.age);

        function innerFunc() {

            // this refers to the global object
            console.log(this.age);
            console.log(this);
        }

        innerFunc();
    }
}

let x = new Person();
x.sayName();
```

## Output



JavaScript

```
25
undefined
Window {}
```

Here, `this.age` inside `this.sayName()` is accessible because `this.sayName()` is the method of an object.

However, `innerFunc()` is a normal function and `this.age` is not accessible because `this` refers to the global object (Window object in the browser).

Hence, `this.age` inside the `innerFunc()` function gives `undefined`.

## Inside an arrow function

```
function Person() {
  this.name = 'Jack',
  this.age = 25,
  this.sayName = function () {

    console.log(this.age);
    let innerFunc = () => {
      console.log(this.age);
    }

    innerFunc();
  }
}
```

```
const x = new Person();
x.sayName();
```

## Output

```
25
25
```

Here, the `innerFunc()` function is defined using the arrow function. And inside the arrow function, `this` refers to the parent's scope. Hence, `this.age` gives **25**.

## Arguments Binding

Regular functions have arguments binding. That's why when you pass arguments to a regular function, you can access them using the `arguments` keyword. For example,

```
let x = function () {  
    console.log(arguments);  
}  
x(4,6,7); // Arguments [4, 6, 7]
```

Arrow functions do not have arguments binding.

When you try to access an argument using the arrow function, it will give an error. For example,

```
let x = () => {  
    console.log(arguments);  
}  
  
x(4,6,7);  
// ReferenceError: Can't find variable: arguments
```

To solve this issue, you can use the [spread](#) syntax. For example,

```
let x = (...n) => {  
    console.log(n);  
}  
  
x(4,6,7); // [4, 6, 7]
```

## Arrow Function with Promises and Callbacks

Arrow functions provide better syntax to write promises and callbacks. For example,

```
// ES5  
asyncFunction().then(function() {  
    return asyncFunction1();  
}).then(function() {
```

```
        return asyncFunction2();
    }).then(function() {
        finish;
    });
```

can be written as

```
// ES6
asyncFunction()
.then(() => asyncFunction1())
.then(() => asyncFunction2())
.then(() => finish);
```

## Things You Should Avoid With Arrow Functions

### 1. You should not use arrow functions to create methods inside objects.

```
let person = {
    name: 'Jack',
    age: 25,
    sayName: () => {

        // this refers to the global .....
        //
        console.log(this.age);
    }
}
```

```
person.sayName(); // undefined
```

[Run Code](#)

### 2. You cannot use an arrow function as a constructor. For example,

```
let Foo = () => {};
let foo = new Foo(); // TypeError: Foo is not a constructor
```

[Run Code](#)

**Note:** Arrow functions were introduced in **ES6**.

## H. How to use different array methods with arrow functions ?

Certainly! Here are some examples of how you can use different array methods with arrow functions:

### 1. **forEach():**

```
const numbers = [1, 2, 3, 4, 5];  
  
numbers.forEach((num) => {  
  
    console.log(num);  
  
});  
  
// Output: 1 2 3 4 5
```

### 2. **map():**

```
const numbers = [1, 2, 3, 4, 5];  
const doubledNumbers = numbers.map((num) => num * 2);  
console.log(doubledNumbers);  
// Output: [2, 4, 6, 8, 10]
```

### 3. **filter():**

```
const numbers = [1, 2, 3, 4, 5];  
const evenNumbers = numbers.filter((num) => num % 2 === 0);  
  
console.log(evenNumbers);  
// Output: [2, 4]
```

### 4. **find():**

```
const numbers = [1, 2, 3, 4, 5];  
const firstEvenNumber = numbers.find((num) => num % 2 === 0);  
  
console.log(firstEvenNumber);  
// Output: 2
```

### 5. **reduce():**

```
const numbers = [1, 2, 3, 4, 5];  
const sum = numbers.reduce((accumulator, num) => accumulator + num, 0);  
  
console.log(sum);  
// Output: 15
```

Note: Similarly, we can use arrow function with all other methods like String methods, Object methods etc.

## 2. String Methods:

- **split():** Splits a string into an array of substrings based on a specified separator.

## JavaScript

- **join()**: Joins all elements of an array into a string.
- **charAt()**: Returns the character at a specified index in a string.
- **toLowerCase()**: Converts a string to lowercase.
- **toUpperCase()**: Converts a string to uppercase.

### 3. Object Methods:

- **hasOwnProperty()**: Returns a boolean indicating whether an object has a specific property.
- **keys()**: Returns an array of a given object's own enumerable property names.
- **values()**: Returns an array of a given object's own enumerable property values.
- **assign()**: Copies the values of all enumerable properties from one or more source objects to a target object.

### 4. Math Methods:

- **random()**: Returns a random number between 0 and 1.
- **floor()**: Rounds a number down to the nearest integer.
- **ceil()**: Rounds a number up to the nearest integer.
- **round()**: Rounds a number to the nearest integer.

### 5. Date Methods:

- **getFullYear()**: Returns the year of a Date object as a four-digit number.
- **getMonth()**: Returns the month of a Date object as a zero-based index (0-11).
- **getDate()**: Returns the day of the month of a Date object (1-31).
- **toISOString()**: Returns a string representing a date in ISO format.

## I. JSON in JavaScript

- JSON (JavaScript Object Notation) is a lightweight data interchange format that is widely used for storing and transmitting data between a server and a client, or between different parts of an application. In JavaScript, JSON is represented as a string and is based on a subset of the JavaScript object literal syntax.
- JSON provides a simple and standardized way to represent structured data, such as objects and arrays, using key-value pairs and ordered lists. It is language-independent, which means it can be used with various programming languages, not just JavaScript.
- JSON data structures are typically used for data serialization and deserialization. Serialization refers to converting data objects into a JSON string, while deserialization involves parsing a JSON string and recreating the original data object.

## JavaScript

Here's an example of a JSON object representing a person's information:

```
{
  "name": "John Doe",
  "age": 30,
  "email": "john.doe@example.com",
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "country": "USA"
  },
  "hobbies": ["reading", "gaming", "traveling"]
}
```

In this example, the JSON object contains various properties such as "name," "age," "email," "address," and "hobbies." Some values are primitive types like strings and numbers, while others are nested objects or arrays.

JavaScript provides built-in methods for working with JSON. The **JSON.parse()** method converts a JSON string into a JavaScript object, and the **JSON.stringify()** method converts a JavaScript object into a JSON string.

```
const jsonString = '{"name":"John Doe","age":30,"email":"john.doe@example.com"}';
const person = JSON.parse(jsonString);
console.log(person.name); // Output: John Doe
const personObject = {
  name: 'Jane Smith',
  age: 25,
  email: 'jane.smith@example.com'
};
```

## JavaScript

```
const personJsonString = JSON.stringify(personObject);  
console.log(personJsonString);  
  
// Output: {"name":"Jane Smith","age":25,"email":"jane.smith@example.com"}
```

JSON is commonly used for exchanging data between a client and a server in web applications, as well as for storing configuration files and transmitting data over HTTP in APIs (Application Programming Interfaces).

## J. localStorage with example

- The localStorage object stores data with no expiration date
- Different methods used in localStorage are: **setItem(key, value)**, **getItem(key)**, **removeItem(key)**, **clear()**, **length**, **key(index)**,

### Save Data to Local Storage

```
localStorage.setItem(key, value);
```

### Read Data from Local Storage

```
let lastname = localStorage.getItem(key);
```

### Remove Data from Local Storage

```
localStorage.removeItem(key);
```

### Remove All (Clear Local Storage)

```
localStorage.clear();
```

## Here's an example of how to use localStorage in JavaScript:

```
// Storing data in localStorage  
localStorage.setItem('name', 'John Doe');  
localStorage.setItem('age', '30');  
  
// Retrieving data from localStorage  
const name = localStorage.getItem('name');
```

## JavaScript

```
const age = localStorage.getItem('age');
```

```
console.log(name); // Output: John Doe
```

```
console.log(age); // Output: 30
```

In the example above, we use **localStorage.setItem(key, value)** to store data with a specified key-value pair. The data is associated with the provided key. Later, we retrieve the data using **localStorage.getItem(key)** by passing the corresponding key.

Note that **localStorage** stores data as strings. If you need to store non-string values like objects or arrays, you can convert them to strings using **JSON.stringify()** before storing and then parse them back to their original format using **JSON.parse()** when retrieving.

```
const user = { name: 'John Doe', age: 30 };
```

```
localStorage.setItem('user', JSON.stringify(user));
```

```
const storedUser = JSON.parse(localStorage.getItem('user'));
```

```
console.log(storedUser.name); // Output: John Doe
```

```
console.log(storedUser.age); // Output: 30
```

## Example: Perform CRUD operation using localStorage methods

Certainly! Here's an example demonstrating CRUD operations on employee data using **localStorage** methods:

```
// Create - Add a new employee to localStorage
const createEmployee = (employee) => {
  const employees = getAllEmployees();
  employees.push(employee);
  localStorage.setItem('employees', JSON.stringify(employees));
};
```

```
// Read - Retrieve all employees from localStorage
const getAllEmployees = () => {
  const employeesString = localStorage.getItem('employees');
  return employeesString ? JSON.parse(employeesString) : [];
};
```

```
// Update - Update an employee in localStorage
const updateEmployee = (id, updatedEmployee) => {
  const employees = getAllEmployees();
```



## JavaScript

```
const updatedEmployees = employees.map((employee) => {
  if (employee.id === id) {
    return { ...employee, ...updatedEmployee };
  }
  return employee;
});
localStorage.setItem('employees', JSON.stringify(updatedEmployees));
};

// Delete - Remove an employee from localStorage
const deleteEmployee = (id) => {
  const employees = getAllEmployees();
  const filteredEmployees = employees.filter((employee) => employee.id !== id);
  localStorage.setItem('employees', JSON.stringify(filteredEmployees));
};

// Usage example
createEmployee({ id: 1, name: 'John Doe', age: 30, department: 'IT' });
createEmployee({ id: 2, name: 'Jane Smith', age: 25, department: 'HR' });
createEmployee({ id: 3, name: 'Tom Johnson', age: 35, department: 'Finance' });

const employees = getAllEmployees();
console.log(employees);
// Output: [{ id: 1, name: 'John Doe', age: 30, department: 'IT' },
//          { id: 2, name: 'Jane Smith', age: 25, department: 'HR' },
//          { id: 3, name: 'Tom Johnson', age: 35, department: 'Finance' }]

updateEmployee(2, { name: 'Jane Brown' });

const updatedEmployees = getAllEmployees();
console.log(updatedEmployees);
// Output: [{ id: 1, name: 'John Doe', age: 30, department: 'IT' },
//          { id: 2, name: 'Jane Brown', age: 25, department: 'HR' },
//          { id: 3, name: 'Tom Johnson', age: 35, department: 'Finance' }]

deleteEmployee(1);

const remainingEmployees = getAllEmployees();
console.log(remainingEmployees);
// Output: [{ id: 2, name: 'Jane Brown', age: 25, department: 'HR' },
```

## JavaScript

```
//      { id: 3, name: 'Tom Johnson', age: 35, department: 'Finance' }]
```

In this example, we define functions for CRUD operations on employee data. The **createEmployee()** function adds a new employee to **localStorage**. The **getAllEmployees()** function retrieves all employees from **localStorage**. The **updateEmployee()** function updates an employee based on the provided ID. The **deleteEmployee()** function removes an employee based on the provided ID.

By using these functions, you can perform CRUD operations on employee data stored in **localStorage**.

## k. Synchronous and Asynchronous in JavaScript

**Synchronous JavaScript:** As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

Let us understand this with the help of an example.

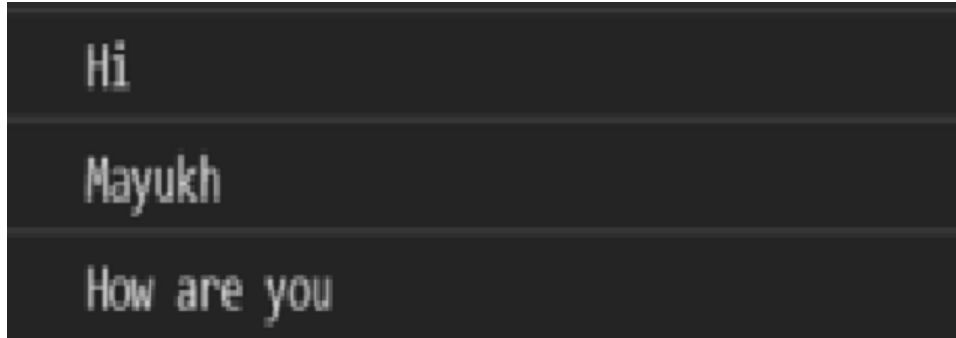
### Example:

```
document.write("Hi"); // First
document.write("<br>");

document.write("Mayukh") ;// Second
document.write("<br>");

document.write("How are you"); // Third
```

### Output:



```
Hi
Mayukh
How are you
```

**Asynchronous JavaScript:** Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

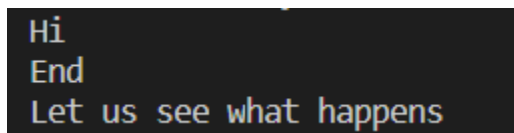
Let us see the example how Asynchronous JavaScript runs.

```
document.write("Hi");
document.write("<br>");

setTimeout(() => {
    document.write("Let us see what happens");
}, 2000);

document.write("<br>");
document.write("End");
document.write("<br>");
```

**Output:**



```
Hi
End
Let us see what happens
```

So, what the code does is first it logs in **Hi** then rather than executing the **setTimeout** function it logs in **End** and then it runs the **setTimeout** function.

At first, as usual, the **Hi** statement got logged in. As we use browsers to run JavaScript, there are the web APIs that handle these things for users. So, what JavaScript does is, it passes the **setTimeout** function in such web API and then we keep on running our code as usual. So it does not block the rest of the code from

executing and after all the code its execution, it gets pushed to the call stack and then finally gets executed. This is what happens in asynchronous JavaScript.

## L. array Vs object in javascript

In JavaScript, both arrays and objects are used to store and organize data, but they have distinct characteristics and purposes.

Array:

- An array is an ordered collection of values.
- It uses integer indices to access and retrieve elements.
- The elements in an array are ordered and can be accessed by their numeric index starting from 0.
- Arrays are represented using square brackets `[]`.
- Arrays are ideal for storing and manipulating lists or collections of similar data.
- Array elements can be of any data type, including numbers, strings, objects, and even other arrays.
- Arrays have built-in methods such as `push()`, `pop()`, `forEach()`, `map()`, and more for performing various operations on the elements.

Example:

```
const numbers = [1, 2, 3, 4, 5];  
console.log(numbers[0]); // Output: 1  
console.log(numbers.length); // Output: 5  
numbers.push(6); // Add 6 to the end of the array
```

Object:

- An object is an unordered collection of key-value pairs.
- It uses keys (strings) to access and retrieve values associated with them.
- The order of properties in an object is not guaranteed.
- Objects are represented using curly braces `{}`.
- Objects are suitable for storing structured data with different properties and values.
- Property keys are unique within an object, and accessing values is done using the corresponding key.
- Object values can be of any data type, including numbers, strings, arrays, other objects, and functions.

- Objects can also have methods, which are functions associated with the object.

Example:

```
const person = {  
  name: 'John Doe',  
  age: 30,  
  city: 'New York',  
};  
console.log(person.name); // Output: John Doe  
console.log(person.age); // Output: 30  
person.email = 'john.doe@example.com'; // Add a new property to the object
```

- In summary, arrays are ordered collections accessed by numeric indices, while objects are unordered collections accessed by keys.
- Arrays are used for lists and collections, while objects are used for storing structured data with properties and values.
- Both arrays and objects have their own unique features and use cases, and they can be nested within each other to create complex data structures.

## ★ Accessing Nested Arrays

objects can contain both nested objects and nested arrays. Similar to accessing nested objects, array bracket notation can be chained to access nested arrays.

Here is an example of how to access a nested array:

```
const ourPets = [  
  {  
    animalType: "cat",  
    names: [  
      "Meowzer",  
      "Fluffy",  
      "Kit-Cat"  
    ]  
  },  
  {  
    animalType: "dog",  
    names: [  
      "Spot",  
      "Buddy"  
    ]  
  }  
];
```

## JavaScript

```
        "Bowser",  
        "Frankie"  
    ]  
}  
];  
  
ourPets[0].names[1];  
ourPets[1].names[0];
```

ourPets[0].names[1] would be the string Fluffy, and ourPets[1].names[0] would be the string Spot.

### ★ how to Nested array and object with each other, Give example

Certainly! You can nest arrays and objects within each other to create complex data structures in JavaScript. Here's an example that demonstrates nesting arrays and objects:

```
const employee1 = {  
  name: 'John Doe',  
  age: 30,  
  departments: ['IT', 'HR'],  
  projects: [  
    { title: 'Project A', status: 'In Progress' },  
    { title: 'Project B', status: 'Completed' },  
  ],  
};
```

```
const employee2 = {  
  name: 'Jane Smith',  
  age: 35,  
  departments: ['Finance'],  
  projects: [  
    { title: 'Project C', status: 'In Progress' },  
    { title: 'Project D', status: 'On Hold' },  
  ],  
};
```

```
const company = {  
  name: 'ABC Corporation',
```

## JavaScript

```
employees: [employee1, employee2],  
};  
  
console.log(company.name); // Output: ABC Corporation  
console.log(company.employees[0].name); // Output: John Doe  
console.log(company.employees[1].projects[1].status); // Output: On Hold
```

In the example above, we have two employee objects, **employee1** and **employee2**, which are then nested within the **company** object. Each employee object contains properties like name, age, departments, and projects. The **departments** property is an array, and the **projects** property is an array of objects.

By nesting arrays and objects, you can create hierarchical data structures that can represent complex relationships and information. In this example, we can access and manipulate the data using dot notation and bracket notation to navigate through the nested structure.

## Convert the above example program using localStorage

Certainly! Here's an example that converts the previous program using **localStorage** to store the nested array and object data:

```
// Create employees and company objects  
const employee1 = {  
  name: 'John Doe',  
  age: 30,  
  departments: ['IT', 'HR'],  
  projects: [  
    { title: 'Project A', status: 'In Progress' },  
    { title: 'Project B', status: 'Completed' },  
  ],  
};  
  
const employee2 = {  
  name: 'Jane Smith',  
  age: 35,  
  departments: ['Finance'],  
  projects: [  
    { title: 'Project C', status: 'In Progress' },  
    { title: 'Project D', status: 'On Hold' },  
  ],  
};
```

## JavaScript

```
    ],  
  };  
  
  const company = {  
    name: 'ABC Corporation',  
    employees: [employee1, employee2],  
  };  
  
  // Store the company object in localStorage  
  localStorage.setItem('company', JSON.stringify(company));  
  
  // Retrieve the company object from localStorage  
  const storedCompany = JSON.parse(localStorage.getItem('company'));  
  
  console.log(storedCompany.name); // Output: ABC Corporation  
  console.log(storedCompany.employees[0].name); // Output: John Doe  
  console.log(storedCompany.employees[1].projects[1].status); // Output: On Hold
```

In this example, we use **localStorage.setItem(key, value)** to store the **company** object by converting it to a JSON string using **JSON.stringify()**. We retrieve the **company** object from **localStorage** using **localStorage.getItem(key)** and parse it back into a JavaScript object using **JSON.parse()**.

By utilizing **localStorage**, we can persist the nested array and object data even if the browser is closed and reopened.

## Spread operator (3 dot)

<https://www.programiz.com/javascript/spread-operator>

```
let arr1 = [0, 1, 2];  
const arr2 = [2, 4, 5];  
  
arr1 = [...arr1, ...arr2];  
console.log(arr1);
```

Output

[0, 1, 2, 2, 4, 5]



## JavaScript

1. Prove that Angular is both client side and server-side programming language with an example?

do we need to always mention return in JavaScript function

Not necessary. If you do not specify a return statement, JavaScript automatically returns undefined. A function does not have to return anything unless you want to. Return can be used to get a value if you need it or return other function and use that function in your code for example.

VVI: <https://mindsers.blog/post/mastering-return-statement/>