

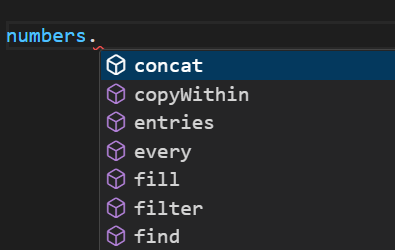
**Adding Elements**:

Let us start by declaring a constant called *numbers* and set it to an array of two elements.

const numbers = [3, 4];

Note that this variable *numbers* is constant so we cannot redefine another variable with same name, but we can modify the content of this array, like adding new elements or remove existing ones.

Since arrays are objects, so using dot operator we can look at the properties and methods in arrays,



*Add element at the end of array🡪push*:

const numbers = [3, 4];

numbers.push(5, 6);

console.log(numbers); //[ 3, 4, 5, 6 ]

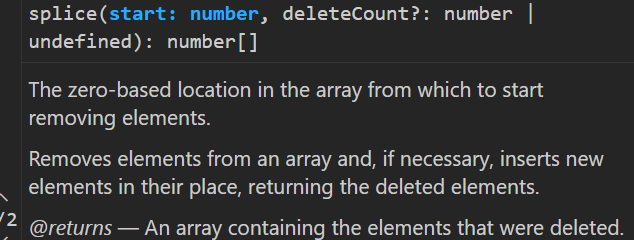
*Add element at the beginning of array 🡪unshift*:

*unshift* method basically pushes the existing elements to the right and adds new elements in the beginning.

numbers.unshift(1, 2);

console.log(numbers); // [ 1, 2, 3, 4, 5, 6 ]

*Add/remove elements from any index of array🡪splice*:



The first parameter is ‘*start’* which is a number and our starting position.

Suppose we need to add a new element between 2 and 3

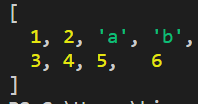
[ 1, 2, 3, 4, 5, 6 ]. So our starting index is 2.

Second argument is ‘*deleteCount’* , how many numbers we want to delete. If none then input 0.

Third argument is an item that we want to add, to stand out we will add two characters ‘a’ and ‘b’,

numbers.splice(2, 0, "a", "b");

console.log(numbers)

🡨 O/P

Note that *a* and *b* are placed after 1 and 2.

**Finding Elements primitives**:

Let us see how can we find elements in an array.

**“***Finding elements in an array really depends on whether you are storing* ***primitive*** *or* ***reference*** *types in an array****”***.

Consider this array with primitive types,

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

We use *indexOf* method, where we pass element that we want to find. If element is found it returns its index if not then it returns -1(*negative one*).

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

console.log(numbers.indexOf("a")); //-1

console.log(numbers.indexOf(1)); // 0

Similar to *indexOf* we have another method called *lastIndexOf* and that will return the last index of the given element.

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

console.log(numbers.lastIndexOf(1)); //3

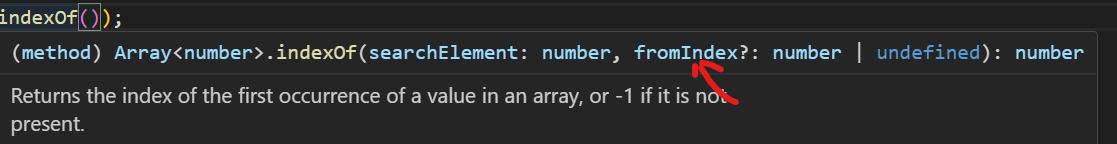
*To check if an element exists in array🡪includes*:

It will return true if element exists.

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

console.log(numbers.includes(1)); //true

All these methods have an optional additional parameter which is *fromIndex*.



Here we will input the index from where we will begin our search.

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

console.log(numbers.indexOf(1)); //0

console.log(numbers.indexOf(1, 2)); //3

**Finding Elements(Objects)**:

Consider this *courses* array of objects,

const courses = [

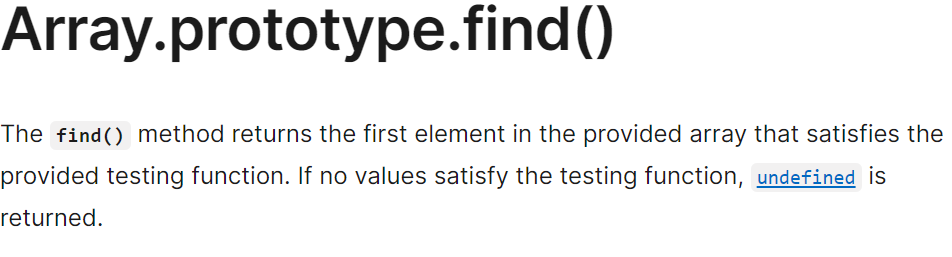
  { id: 1, name: "a" },

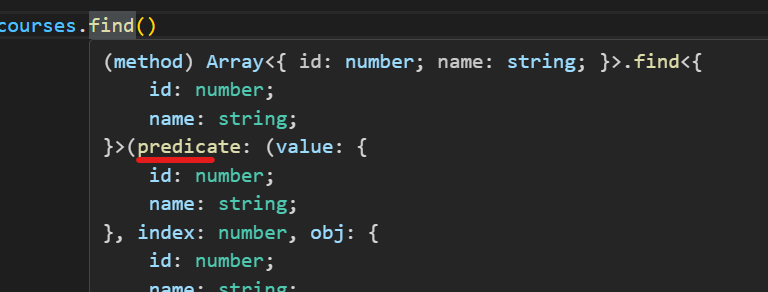
  { id: 2, name: "b" },

];

Let us see if have a course with name “a”.

Our *includes* method will not work here, we will have to use *find* method.





As an argument to this find function , we need to pass a function and we call this function a ***predicate*** and we use it to determine if an element exists in an array or not.

***“****As a parameter to this predicate function, we pass an element and in the body we return a Boolean expression****”***

const course = courses.find(function (course) {

  return course.name === "a";

});

console.log(course); //{ id: 1, name: 'a' }

If there are no element matching the given criteria we get undefined.

const course = courses.find(function (course) {

  return course.name === "c";

});

console.log(course); //undefined

Note: *find* method returns the first element that matches our criteria. We have a similar method called *findIndex* but instead of returning the actual element it returns the index of that element.

const course = courses.findIndex(function (course) {

  return course.name === "b";

});

console.log(course); //1

**Arrow Functions**:

In the last lecture you learned about the predicate function that we passed to the find method. In ES6 there is shorter and cleaner way to write the same code using *arrow* functions.

***“****Whenever you want to pass a function as a callback function as an argument for a different method, you can use the arrow function syntax****”***

const courses = [

  { id: 1, name: "a" },

  { id: 2, name: "b" },

];

const course = courses.find((course) => {

  return course.name === "a";

});

console.log(course); //{ id: 1, name: 'a' }

Here we removed *function* keyword and to separate parameters of this function from its body we put a fat arrow in between them, this is what we call an *arrow function*.

Note: If your arrow function has a single parameter we can even remove parentheses and we can put an empty parentheses in case we have no parameter.

In case we are returning just a single value, we can make our code even shorter.

const course = courses.find((course) => course.name === "a");

console.log(course); //{ id: 1, name: 'a' }

Here we got rid of return keyword and removed curly braces enclosing function body, still getting the same result.

We can read the above expression as ,

*course goes to course.name equals ‘a’, so basically we are finding a course with the name equal to ‘a’*.

**Removing elements from array**:

*Remove last element🡪pop*:

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

numbers.pop();

console.log(numbers); //[ 1, 2, 3 ]

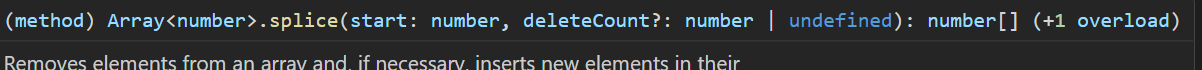
*Remove first element🡪shift*:

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

numbers.shift();

console.log(numbers); //[ 2, 3, 4 ]

*Remove middle element 🡪splice*:



This splice method takes two arguments, start index from where we want to start deleting and deleteCount means how many elements we want to delete.

For this example, first let us delete only one number which is at index 2.

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

numbers.splice(2, 1);

console.log(numbers); //[ 1, 2, 4 ]

only 3 is deleted, we can delete multiple numbers as well.

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

numbers.splice(2, 2);

console.log(numbers); //[ 1, 2 ]

**Emptying an array**:

In this section we will learn how to remove all the items from an array, There are a bunch of solutions for same.

*🡪 Reassign original array to a new array*:

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

numbers = [];

console.log(numbers); //[]

But note that array should not be declared as a *const* in first place.

*Also it does not work if you have* ***multiple references*** *to this array*.

Because this old array is still in memory, so if there are any references to this array eventually it will be removed by the *garbage collector*.

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

let another = numbers;

So if we have *another* variable which is also pointing to the same object, the original array will not be garbage collected.

So when numbers = [ ] , *numbers* is now pointing to a new object in memory but *another* is still pointing to old object.

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

let another = numbers;

numbers = [];

console.log(numbers); //[] 🡪new object

console.log(another); //[1, 2, 3, 4] 🡪still old object

So if you have multiple references to the original array, then you need to use other solutions.

*Assigning length of array to 0(****recommended****)*:

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

let another = numbers;

numbers.length = 0;

console.log(numbers); //[]

console.log(another); //[]

As you can see original array is empty now as well as its reference.

*Using splice method*:

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

let another = numbers;

numbers.splice(0, numbers.length);

console.log(numbers); //[]

console.log(another); //[]

*Using pop method with loop*(*not recommended*):

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

let another = numbers;

while (numbers.length > 0) {

  numbers.pop();

}

console.log(numbers); //[]

console.log(another); //[]

**Combining Arrays**:

Let us see how can we combine two arrays or slice an array into two parts.

const first = [1, 2, 3];

const second = [4, 5, 6];

To *combine these two arrays* we use *concat* method,

It will return a new array which is a combination of first and second array.

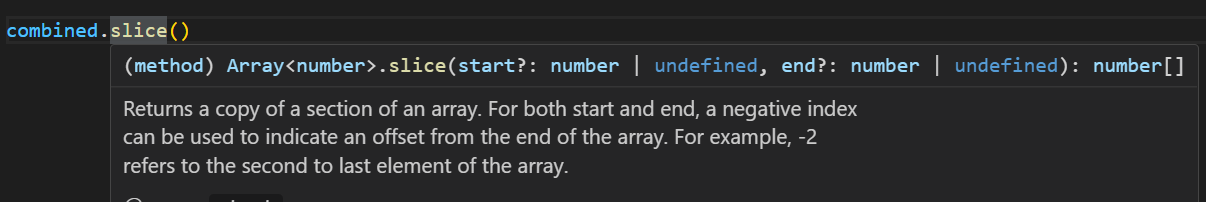
const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

console.log(combined); // [ 1, 2, 3, 4, 5, 6 ]

We have another method which is opposite of *concat*, it is *slice* and used to *slice an array into two parts*.



Now there are multiple ways to use this method one way is to use *start* and *end* index.

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = first.concat(second);

console.log(combined); // [ 1, 2, 3, 4, 5, 6 ]

const sliced = combined.slice(2, 4);

console.log(sliced); //[3, 4]

*If we exclude the last index argument, we get all the elements after start index*,

const sliced = combined.slice(2);

console.log(sliced); //[ 3, 4, 5, 6 ]

*What if we concat/slice the arrays which has reference type*?

In case of primitives, the primitive value is copied into target array but *if you have objects in your array, objects themselves are not copied only their references are copied*.

const first = [{ id: 1 }];

const second = [4, 5, 6];

const combined = first.concat(second);

When we call the *concat* method, this {*id: 1*}object itself is not copied to the *combined* array only its reference is copied.

const first = [{ id: 1 }];

const second = [4, 5, 6];

const combined = first.concat(second);

first[0].id = 10; //change the value of id to 10

console.log(combined); //[ { id: 10 }, 4, 5, 6 ]

So we can see in the *combined* array, id is changed to 10 as well. Since we are dealing with a reference type, the object is copied by its reference. Same principal is applied to the slice method.

**The spread operator**:

So in the last lecture we learned how to use concat method to combine two arrays. There is another way to achieve the same thing using the *spread* operator.

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = [...first, ...second];

console.log(combined); //[ 1, 2, 3, 4, 5, 6 ]

When we use spread (…) operator like this all its elements are spread individually.

So basically we are declaring a new array and in that array we are adding the individual elements of the first and second arrays.

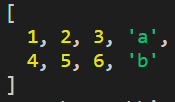
We have more flexibility while using spread operator,

const first = [1, 2, 3];

const second = [4, 5, 6];

const combined = [...first, "a", ...second, "b"];

console.log(combined);

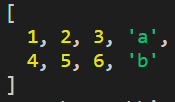
🡨We can even add new elements between the combined arrays or at start or end.

We can even copy an entire array,

const combined = [...first, "a", ...second, "b"];

const copy = [...combined];

console.log(copy);



**Iterating an Array**:

*Using for…of loop*,

const numbers = [1, 2, 3];

for (number of numbers) {

  console.log(number); //1 2 3

}

Using *forEach* method(*most commonly used*),

Each array has this *forEach* method, that takes a callback function.

const numbers = [1, 2, 3];

numbers.forEach((number) => console.log(number)); //1 2 3

When we use *forEach* method, *the callback function is executed for each element in the array. Each element will be passed as an argument to this function*.

Note: The callback function of *forEach*, takes an additional parameter which is the index.

numbers.forEach((number, index) => console.log(index, number));

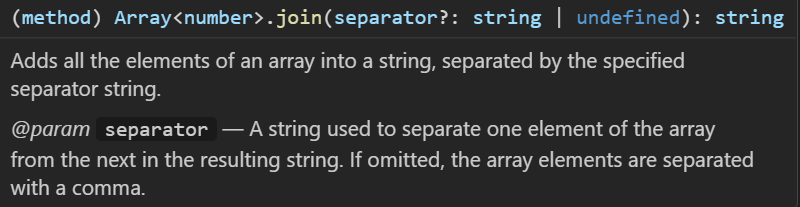


**Joining Arrays**:

Another useful method of arrays is *join*. For example in this numbers array,

const numbers = [1, 2, 3];

Let us say we want to join its elements, so we use *join* method.



Note: See the parameter here which is *separator****?***, when you see a question mark with a parameter means it is optional. So *separator?: string* means we can optionally pass a string here.

Also :string at the end means that this method returns a string.

const numbers = [1, 2, 3];

const joined = numbers.join(",");

console.log(joined); // 1,2,3

All our elements are joined together by comma.

Another useful method that goes hand in hand with join is *split* method but it applies on string.

const message = "This is my first message";

Let us say we have a *message* string.

We can split the *message* like this,

const message = "This is my first message";

const parts = message.split(" "); //separate strings with whitespace

console.log(parts); // [ 'This', 'is', 'my', 'first', 'message' ]

We get an array of strings.

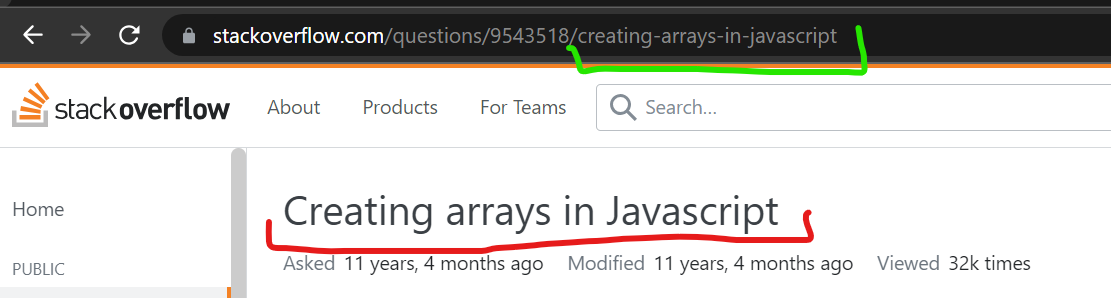
Now we can apply join method to combine these elements using a hyphen as a separator and get a string.

const joined = parts.join("-");

console.log(joined); // This-is-my-first-message

***“****This technique of using* ***split*** *and* ***join*** *methods together like this, is particularly useful when* ***building a URL slug”***.

For example, Look at the title of the page and URL



*creating arrays in Javascript* is the title which user searched on google but in URL we see *creating-arrays-in-javascript*.

So the title is converted to url slug when all the whitespaces are replaced with a hyphen.

**Sorting Arrays**:

Here we have an array of numbers in unsorted order.

const numbers = [2, 3, 1];

*To sort this in ascending order 🡪 sort* :

numbers.sort();

console.log(numbers); // [ 1, 2, 3 ]

To sort in descending order 🡪 *reverse*:

**Not working as expected, numbers not reversing**.

Letters like “a”, “b” and “c” are reversing as expected but not numbers.

**Solution**: To solve this problem, *First sort () then reverse (),* DO NOT USE REVERSE ALONE.

*Sorting objects*:

const courses = [

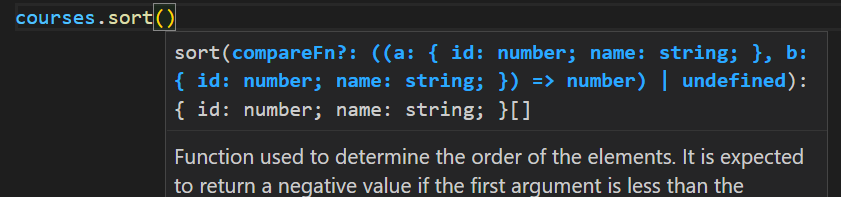
  { id: 1, name: "Node.js" },

  { id: 2, name: "JavaScript" },

];

We want to sort this array of courses by name of the course.

The sort method takes an optional argument which is a function.



This function is used for comparison and takes two parameters *a* and *b*,

courses.sort(function(a,b){

})

If a < b = -1

a > b = 1

a = b = 0

const courses = [

  { id: 1, name: "Node.js" },

  { id: 2, name: "JavaScript" },

];

courses.sort(function (a, b) {

  if (a.name < b.name) return -1;

  if (a.name > b.name) return 1;

  return 0;

});

console.log(courses);

// [{ id: 2, name: 'JavaScript' }, { id: 1, name: 'Node.js' }]

Courses are sorted now as per the name but if I change JavaScript to lowercase javaScript then Node.js will come in the beginning.

It is because each character is internally represented by a number. Look at the ASCII (*American standard code for information exchange*)table.

Lowercase j is 106 and uppercase N is 78, so Node.js came first while sorting.

So to fix this issue while sorting, we should remove case sensitivity,

const courses = [

  { id: 1, name: "Node.js" },

  { id: 2, name: "javaScript" },

];

courses.sort(function (a, b) {

  const nameA = a.name.toUpperCase();

  const nameB = b.name.toUpperCase();

  if (nameA < nameB) return -1;

  if (nameA > nameB) return 1;

  return 0;

});

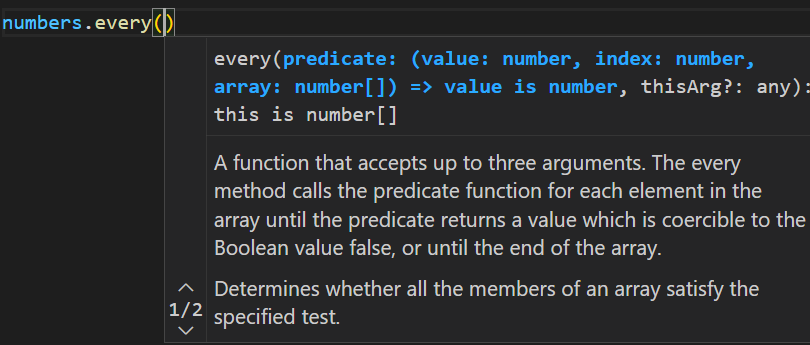
console.log(courses); [{ id: 2, name: 'javaScript' }, { id: 1, name: 'Node.js' }]

**Testing the elements of an Array**:

We have two new methods in modern JavaScript called *every* and *some*,

So here we have a numbers array and we want to check to see if all numbers in this array are positive.

const numbers = [1, 2, 3];



This every method takes a callback function which takes three parameters *value*, *index* and *array* itself. Note that not all parameters are required.

const numbers = [1, 2, 3];

allPositives = numbers.every(function (value) {

  return value >= 0; //check if value is positive

});

console.log(allPositives); //true

In case any value is negative,

const numbers = [1, -1, 2, 3];

allPositives = numbers.every(function (value) {

  return value >= 0;

});

console.log(allPositives); //false

***“****This callback function in* ***every*** *method runs on every element of array and as soon as it finds an element that does not meet its criteria, It stops searching****”***.

The other method is *some* and that *checks to see if we have* ***at least one*** *element in this array which matches our criteria*.

const numbers = [1, -1, 2, 3];

const atleastOnePositive = numbers.some(function (value) {

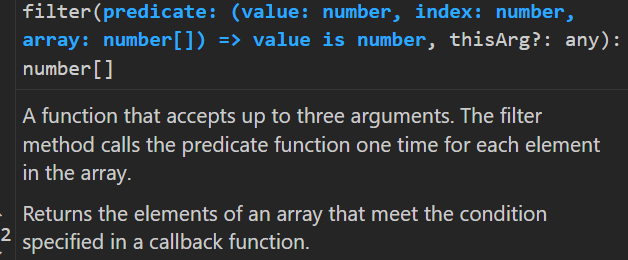
  return value >= 0;

});

console.log(atleastOnePositive); //true

**Filtering an array**:

Let us return numbers greater than equal to 2, so we can use filter method,



const numbers = [1, -1, 2, 3];

const filteredArray = numbers.filter((number) => number >= 2);

console.log(filteredArray); //[ 2, 3 ]

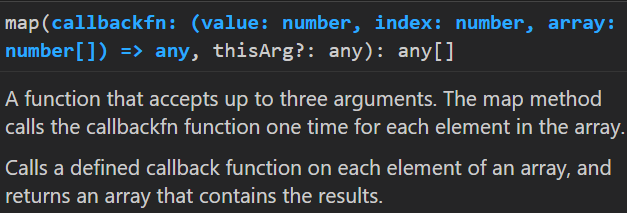
***“****In a real world application, we might have to filter an array of objects like filtering shoes below a price range or showing all restaurants with 4 star rating or above****”***.

**Mapping an Array**:

Another very useful and powerful method we have in modern JavaScript is the *map* method. With this method we can map each item in an array to something else.

const numbers = [1, 2, 3];

map f



Let us say we want to create some HTML markup using this array.

const numbers = [1, 2, 3];

const items = numbers.map((number) => `<li>${number}</li>`);

console.log(items); // [ '<li>1</li>', '<li>2</li>', '<li>3</li>' ]

So we can see each number is mapped to a string, which is our list item.

Now we will use join method to join the elements of this array and create a string.

const numbers = [1, 2, 3];

const items = numbers.map((number) => `<li>${number}</li>`);

const html = items.join("");

console.log(html); // <li>1</li><li>2</li><li>3</li>

Now instead of an array we have a string.

We can concatenate these list items with a *<ul>* (*unordered list*) tag.

const html = "<ul>" + items.join("") + "</ul>";

console.log(html); // <ul><li>1</li><li>2</li><li>3</li></ul>

We can also map elements of an array with an object,

const numbers = [1, 2, 3];

const items = numbers.map((number) => {

  const obj = { value: number };

  return obj;

});

console.log(items); // [ { value: 1 }, { value: 2 }, { value: 3 } ]

Note: In the callback function we do not need to declare a const obj variable. We can simply return it like this,

const numbers = [1, 2, 3];

const items = numbers.map((number) => {

  return { value: number };

});

console.log(items); //[ { value: 1 }, { value: 2 }, { value: 3 } ]

and still get the same result.

Note: Both filter and map method do not modify the original array, they return a new one. Also they are chainable means they can be called one after the other,

Here I will add a *filter* (number >= 2) then immediately map it.

const numbers = [1, 2, 3];

const items = numbers

  .filter((number) => number >= 2)

  .map((number) => {

    return { value: number };

  });

console.log(items); //[ { value: 2 }, { value: 3 } ]

Now *map* function only maps value 2 and 3.

**Reducing an Array**:

Once again with our array of numbers,

const numbers = [1, -1, 2, 3];

Let us say we want to calculate the sum of all these numbers in this array. This will be similar to calculate sum of all items in a shopping cart.

By using *for* loop:

const numbers = [1, -1, 2, 3];

let sum = 0;

for (number of numbers) {

  sum += number;

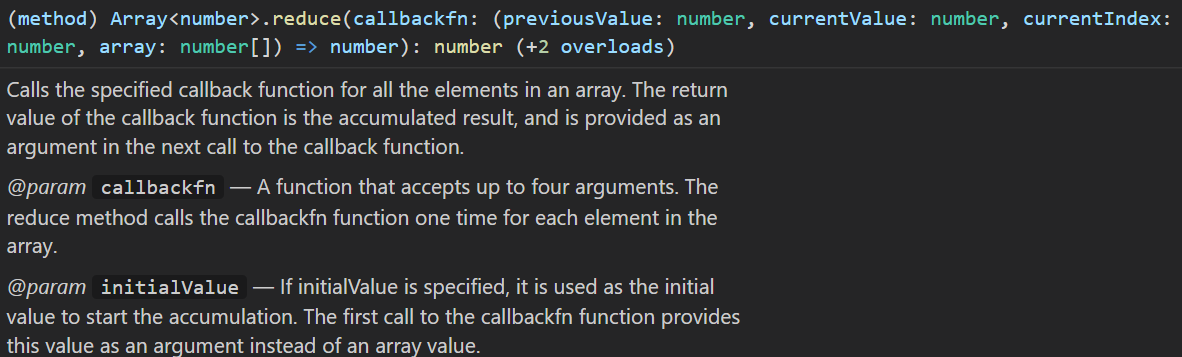
}

console.log(sum); //5

There is shorter and cleaner way to write the same code, which is by using *reduce* method.

***“****By using reduce method we can reduce all the element of an array to a single value, that single value can be a number, string or object etc…****”***

In this example, we want to reduce all the elements of array into a single number, which is the sum of all numbers in this array.



This *reduce* method takes a callback function with two parameters, *accumulator* and *current value*.

numbers.reduce((accumulator, currentValue)=>{

})

This *accumulator* parameter is exactly like *sum* variable that we defined in the previous for loop example.

***“Accumulator*** *is something that we initialize and then callback function is executed multiple times and each time its* ***current value*** *will be set to one element of array*.

In this example, where we want sum of all values in this array, for each call we want to get this current value and add it to accumulator.

const numbers = [1, -1, 2, 3];

numbers.reduce((accumulator, currentValue) => {

  return accumulator + currentValue;

});

Internally, this reduce method will get this result and store it in the accumulator.

Since we need to initialize accumulator to 0 for this example, we will pass 0 as the second argument of the reduce method.

const numbers = [1, -1, 2, 3];

numbers.reduce((accumulator, currentValue) => {

  return accumulator + currentValue;

}, 0);

Note: This reduce method has two arguments, First argument is the callback function and second argument is the initial value for the accumulator.

const numbers = [1, -1, 2, 3];

const sum = numbers.reduce((accumulator, currentValue) => {

  return accumulator + currentValue;

}, 0);

console.log(sum); //5

If no initial value supplied for the accumulator then by default its value will be the first element of the array,

**Exercise1 – Array from Range**:

🡪Give a min to max range to a function and *push* all the elements in that range to an array.

function arrayFromRange(min, max) {

  let elements = [];

  for (let i = min; i <= max; i++) {

    elements.push(i);

  }

  return elements;

}

let result = arrayFromRange(-1, 4);

console.log(result); //[ -1, 0, 1, 2, 3, 4 ]

**Exercise 2 - Includes**:

🡪 Write a function like *includes*, which check to see if an element exist in our array or not.

let another = [1, 2, 3, 4];

function includes(array, searchElement) {

  for (let i = 0; i < array.length; i++) {

    if (array[i] === searchElement) return true;

  }

  return false;

}

console.log(includes(another, 2)); //true

console.log(includes(another, 5)); //false

🡪 Using *for – of* approach

let another = [1, 2, 3, 4];

function includes(array, searchElement) {

  for (let element of array) {

    if (element === searchElement) return true;

  }

  return false;

}

console.log(includes(another, 2)); //true

console.log(includes(another, 5)); //false

**Exercise 3 Except**:

Show only those elements from first array which are not in second array

let array = [1, 2, 3, 4, 1, 1, 1]; //first array

let excluded = [1, 2]; //second array

//use includes and push method

function except(array, excluded) {

  let result = [];

  for (let arr of array) {

    if (!excluded.includes(arr)) {

      result.push(arr);

    }

  }

  return result;

}

console.log(except(array, excluded)); result =[ 3, 4 ]

**Exercise 4 – Moving an Element**:

Write a function called *move*,

Which moves a specific element chosen by its *index* to a different position given as *offset* Also handle the invalid offset.

*Hint: Use splice method*

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

function move(array, index, offset) {

  const position = index + offset;

  if (position >= array.length || position < 0) {

    console.error("Invalid offset");

    return;

  }

  let output = [...array];

  let element = output.splice(index, 1)[0];

  output.splice(index + offset, 0, element);

  if (offset > output.length || 0) return console.error("Invalid Offset");

  return output;

}

console.log(move(numbers, 1, -1)); //[ 2, 1, 3, 4 ]

**Exercise 5 Count the occurrences**:

Find the occurrence of an element inside an array using reduce method

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

function countOccurence(array, searchElement) {

  const occurence = array.reduce(

    (acc, current) => (searchElement == current ? acc + 1 : acc),

    0

  );

  return occurence;

}

console.log(countOccurence(numbers, 3));

**Exercise 6: Get Max**:

Find the largest number out of all array elements. If we pass an empty array return undefined.

*Using for loop*:

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

function getMax(array) {

if (array.length === 0) return undefined;

  let max = array[0];

  for (i = 0; i <= array.length; i++) {

    if (max < array[i]) {

      max = array[i];

    }

  }

  return max;

}

console.log(getMax(numbers));

*Using reduce*:

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

function getMax(array) {

  if (array.length === 0) return undefined;

  let max = array.reduce((acc, current) => (acc < current ? current : acc));

  return max;

}

console.log(getMax(numbers));

**Exercise 7: Movies**:

const movies = [

  { title: "a", year: 2018, rating: 4.5 },

  { title: "b", year: 2018, rating: 4.7 },

  { title: "c", year: 2018, rating: 3 },

  { title: "d", year: 2017, rating: 4.5 },

];

*Print*:

🡪 All movies in 2018 with rating > 4

🡪 Sort them by their rating

🡪 Descending order

🡪 Pick their title

*Expected*: “b”, “a”

const movies = [

  { title: "a", year: 2018, rating: 4.5 },

  { title: "b", year: 2018, rating: 4.7 },

  { title: "c", year: 2018, rating: 3 },

  { title: "d", year: 2017, rating: 4.5 },

];

let titles = movies

  .filter((movie) => movie.rating > 4 && movie.year === 2018)

  .sort((a, b) => a.rating - b.rating)

  .reverse()

  .map((movie) => movie.title);

console.log(titles); [ 'b', 'a' ]