**Array Functions**

**1) push() -** It is used to insert the elements in an array at the nth+1 index if the length of an array is n. And this push can takes n parameters accordingly.

Its return value will be updated length upon which it is used

*Example -* 1

*const animals = ["pigs", "goats", "sheep"]*

*animals.push("cat",*”dog”*)*

*console.log(animals)*

*// result = ["pigs", "goats", "sheep",”cat”,”dog”]*

**Adding element to an array :** If i have an array and i push the items to that array and stores it into another variable then it will gives me new array length

*Example - 2*

const sports = ["soccer", "baseball"]

const total = sports.push("football", "swimming")

console.log(sports) //["soccer", "baseball", "football", "...]

console.log(total); //4

**Merging two arrays :** can be done by this destructuring method

**const vegetables = ["parsnip", "potato"];**

**const moreVegs = ["celery", "beetroot"];**

***// Merge the second array into the first one***

**vegetables.push(...moreVegs);**

**console.log(vegetables); *// ['parsnip', 'potato', 'celery', 'beetroot']***

***We can achieve this by using concat() function too.***

**Push on non-array objects-**

Push method reads the length property of this. The index of this will be set according to the length with the argument passed to the push().

*Example - 3*

const arrayLike = {

length: 3,

unrelated: "foo",

2: 4,

};

Array.prototype.push.call(arrayLike, 1, 2);

console.log(arrayLike);// {2: 4, 3: 1, 4: 2, length: 5, unrelated:”foo”}

const plainObj = {};

Array.prototype.push.call(plainObj, 1, 2);

console.log(plainObj); //{0: 1, 1: 2, length: 2}

*What if i do changes in length see example-*

*const arrayLike = {*

*length: 5,*

*unrelated: "foo",*

*2: 4,*

*};*

*Array.prototype.push.call(arrayLike, 1, 2);*

*console.log(arrayLike);//{2: 4, 5: 1, 6: 2, length: 7, unrelated:”foo”} see the index starts from length 5*

**Objects in array manner:**

Here this points to this object. And we do not use Array.prototype format here.

**const obj = {**

**length:2,**

**addelem(*elem*){**

**[].push.call(this, elem)**

**}**

**}**

**obj.addelem(5)**

**obj.addelem(6)**

**console.log(obj)// {2: 5, 3: 6, length: 4, addelem: add...}**

**2) pop()-** pop in array is used to remove the last elem of the array. And returns the array with the updated length.

Its return value will be element which is removed. If empty then return value will be undefined

const plants = ["broccoli", "cauliflower", "cabbage", "kale", "tomato"]

plants.pop()

console.log(plants)//[ 'broccoli', 'cauliflower', 'cabbage','kale']

**Storing popped element into variables-** this popped will stores the popped element of an array

**const myFish = ["angel", "clown", "mandarin", "sturgeon"];**

**const popped = myFish.pop();**

**console.log(myFish); *// ['angel', 'clown', 'mandarin' ]***

**console.log(popped); *// 'sturgeon'***

**Calling pop() on non-array -** it will gives the popped elem,updatedArr

const arrayLike = {

length: 3,

unrelated: "foo",

2: 4,

};

console.log(Array.prototype.pop.call(arrayLike)) *// 4*

console.log(arrayLike) *// { length: 2, unrelated: 'foo' }*

**Object with an array-** without protopype and with function

**const collection = {**

**lenght:0,**

**addElem(...*elements*) {**

**return [].push.call(this, ...elements);**

**},**

**removeElem() {**

**return [].pop.call(this);**

**},**

**}**

**collection.addElem(10,20,30)**

**console.log(collection.length)**

**collection.removeElem()**

**console.log(collection.length)**

**3) Unshift Function –** it is similar to push function but the only difference is that it adds the elements at the beginning of an array.

It return value is new length upon which it is called

const arr1 = [1,2,3]

console.log(arr1.unshift(4,5)) *// output: 5*

console.log(arr1) *// output : array [4,5,1,2,3]*

If I insert multiple elements together in unshift then I will consider it as a single chunk and insert that chunk in the beginning. Like in above example inserting 4,5 as an single chunk.

But if I insert one by one then the last added will be placed first see example.

const arr1 = [1,2,3]

arr1.unshift(4)

arr1.unshift(5)

arr1.unshift(6)

console.log(arr1) *// output : array [6,5,4,1,2,3]*

*arr1.unshift([8,9]) // output : array [[8,9],6,5,4,1,2,3]*

**Unshift on non-array:**

Unshift method reads length property of this and it shift the value in the range 0 to length-1 and the index will be starts from 0 and length will be previous length plus no of added values

const obj = {

length:3,

unrelated:”foo”,

2:4

}

Array.prototype.unshift.call(obj,1,2)

Console.log(obj) // // {'0':1,'1':2,'4':4,length:5,unrelated:'foo'}

Const plainobj = {}

Array.prototype.unshift.call(plainobj,1,2)

Console.log(plainobj) // { '0': 1, '1': 2, length: 2 }

Without using prototype –

const obj = {

length:0,

addElem(elem){

[].unshift.call(obj,elem) // we can take this instead of obj

}

Obj.addElem(4)

Obj.addElem(5)

Console.log(obj)// *{0: 4, 1: 5, length: 2, addElem: f addElem()}*

**4) Shift function**

**There is shift function which is used same as pop() but the only difference is** shift is used to remove elements form the beginning.

Example of Shift with condition –

Const names = ["Andrew", "Tyrone", "Paul", "Maria", "Gayatri"]

while ((i = names.shift()) !== undefined) {

  console.log(i); // Andrew,Tyrone,Paul,Maria,Gayatri

}

console.log(names) *// empty []*

here this I stores every removed element. And this loop runs untill undefined has occurred.

Type of undefined will be “undefined”.

5) **Reverse function**

Reverse function in an array is used to reverse the array. It reverse the same array and return us the reference of the same array

const array1 = ["one", "two", "three"];

console.log(array1); *//output:Array ["one", "two", "three"]*

const reversed = array1.reverse();

console.log(reversed); //output:Array ["three", "two", ”one”]

console.log(array1);//output:Array ["three", "two", "one"]

this is because this reverse changes the original array also

Reverse takes no parameters and it returns the reference to the original array.

Reversing by mutating element – directly modify the original array

Here both refers to the reversed array because original array is reversed here beacause of this numbers.reverse()

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

const reversed = numbers.reverse();

reversed[0] = 5;

console.log(numbers[0]); // 5

Reversing by non mutating elements – do not directly modify original array we create a shallow copy of the array by using spread(…numbers)

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

const reversed = […numbers].reverse();

reversed[0] = 5;

console.log(numbers[0]); // 3

now this numbers points to the original array and this reversed points to he new copy of the array which is reversed.

Reverse on sparse array – sparse array will be reversed same as above sparse array always remains sparse after using reverse

console.log([1, , 3].reverse()); // [3, empty, 1]

Reverse with non-array:

Reverse method reads the length property of this . This reverse will be applied if the key of elements should have index between 0 and length/2.

If this arrayLike contains 6:4,7:33 then reverse will not be applied and it will prints the same result as arrayLike.

const arrayLike = {

  length: 4,

  unrelated: "foo",

  2: 4,

  3: 33,

};

console.log(Array.prototype.reverse.call(arrayLike));

the output will be *// {0: 33, 1: 4, length: 4, unrelated: ...}*

if the length will be 3 then this 3:33 will be ignored and remains the same.

6) **slice() function -** this function returns a shallow copy it do not changes the original array. It consists of two parameters one is start and the another is end and both are optional.

Slice returns the new array with extrated values form start to end.

START – always be integer. Can be positive or negative both and if start greater then length of array then empy arry will be returned. Start will always be included.

END - always be integer. Can be positive or negative both and end will not be included.

If start will be negative and end will be positive then returns empty array.

const animals = ["ant", "bison", "camel", "duck", "elephant"];

console.log(animals.slice(7)) //output: Array ["camel", "duck", "elephant"]

console.log(animals.slice(2, 4)) //output: Array ["camel", "duck"]

console.log(animals.slice(1, 5)) //output: Array ["bison", "camel", "duck", "elephant"]

console.log(animals.slice(-3,-1)) //output: Array ["duck", "elephant"]

console.log(animals.slice(2, -1)) //output: Array ["camel", "duck"]

console.log(animals.slice()) //output: Array ["ant", "bison", "camel", "duck", "elephant"]

**Using slice with array of objects –** it will behave same as above we have done

const myHonda = {

color: "red",

wheels: 4,

engine: { cylinders: 4, size: 2.2 },

};

const myCar = [myHonda, 2, "cherry condition", "purchased 1997"];

const newCar = myCar.slice(0, 2);

console.log(myCar); // mycar array will be returned

console.log(newCar); // newcar array will ne returned after slice

console.log(myCar[0].color); // it return red

console.log(newCar[0].color); // also return red

// now we are changing color of object

myHonda.color = "purple";

console.log(myHonda.color);// now color will be purple

console.log(myCar[0].color); // color will be purple

console.log(newCar[0].color); //color will be purple because changes will be reflect to shallow copy.

**Slice on non-arrray objects –** the slice method reads the length property of this. And it reads only integer keys only.

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 33, // ignored by slice()because length is 3

};

console.log(Array.prototype.slice.call(arrayLike, 1, 3));

//[3,4]

If I give key in non integer form then it will not accept this it will gives empty item.

const arrayLike = {

length: 5,

0: 2,

1: 3,

name:"chirag",

3: 4,

4: 33, // ignored by slice() since length is 3

};

console.log(Array.prototype.slice.call(arrayLike, 1, 5))

// [3,<empty item>,4,33]

Slice is used to convert array like objects to array. Like I want to convert (1,2,3 ) into array.

function slice(args) {

return Array.prototype.slice.call(args);

}

function list() {

return slice(arguments);

}

const list1 = list(1, 2, 3)

console.log(list1)}; // [1, 2, 3]

if the array will be sparse then will always return array with sparce.

7) **splice() function -**  This function is used to remove the elements or replace the elements and add new elements within the array. It changes the new array.

Splice consist of three parameters which are start,deletecount,items

Start - zero based index tells us form where we have to start.

Deletecount – tells us who many items we have to delete from start till delete count

Items – tells us what to add in an array

Return value will be an array containing delete count.

const months = ["Jan", "March", "April", "June"];

months.splice(1, 0, "Feb"); // Inserts at index 1

console.log(months) //Array ["Jan", "Feb", "March", "April", "June"] this zero means no element is removed

months.splice(4, 1, "May") // Replaces 1 element at index 4

console.log(months) //Array ["Jan", "Feb", "March", "April", "May"]

// remove 0 elements from index-2

const myFish = ["angel", "clown", "mandarin", "sturgeon"];

const removed = myFish.splice(2, 0, "drum");

console.log(removed) // []

console.log(myFish) // ["angel", "clown", "drum", "mandarin", "sturgeon"]

// adding at last

const myFish = ["angel", "clown", "mandarin"];

const removed = myFish.splice(myFish.length, 0, "sturgeon");

console.log(removed) // []

console.log(myFish) // ["angel", "clown", "mandarin", "sturgeon"]

Removing 1 element at index-2 and inserting another example

const myFish = ["angel", "clown", "drum", "sturgeon"];

const removed = myFish.splice(2, 1, "trumpet");

// myFish is ["angel", "clown", "trumpet", "sturgeon"]

// removed is ["drum"]

**Splice on non array objects –**

This method reads the length property of this it then update the key and length property of an object.

const arrayLike = {

length: 3,

unrelated: "foo",

0: 5,

2: 4,

};

console.log(Array.prototype.splice.call(arrayLike, 0, 1, 2, 3));

// [ 5 ]

console.log(arrayLike);

// { '0': 2, '1': 3, '3': 4, length: 4, unrelated: 'foo' }

In this example length is 3 if I replace index of arrayLike 0:5,4:4 then this 4:4 will be excluded because length will include which have index less then 3 and after 1 the index will become 3 because gap between 0 and 2 is of 1 so

**8) Concat() function –** this function is used to merge two array and it do not change the original array it gives us the new array.

const letters = ["a", "b", "c"];

const numbers = [1, 2, 3];

const alphaNumeric = letters.concat(numbers);

console.log(alphaNumeric) // results in ['a', 'b', 'c', 1, 2, 3]

//concating three array

const num1 = [1, 2, 3];

const num2 = [4, 5, 6];

const num3 = [7, 8, 9];

const numbers = num1.concat(num2, num3);

console.log(numbers) // [1, 2, 3, 4, 5, 6, 7, 8, 9]

concatenating to an array by using two examples one direct and another using two variables see difference

const letters = [["a"], "b", "c"];

const alphaNumeric = letters.concat(1, [2, 3]);

console.log(alphaNumeric); //[['a'], 'b', 'c', 1, 2, 3]

in above example direct concatenating is held and it concat 1,2,3 because here you are concatenate three arrays.

const num1 = [[1]];

const num2 = [2, [3]];

const numbers = num1.concat(num2);

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

here you can see the changes in the output it will concate nested arrays.

***Concatenating array-like objects with Symbol.isConcatSpreadable.***

Concat can also be used to merge two object but if I want to change my object to array I can use this Symbol.isConcatSpreadable()

Which I have to make it true to convert into array and it takes or concat if key are integers.

const obj1 = { 0: 1, 1: 2, 2: 3, length: 3 };

const obj2 = { 0: 1, 1: 2, 2: 3, 3:"chirag", length: 4, [Symbol.isConcatSpreadable]: true };

console.log([0].concat(obj1, obj2)) // [ 0, { '0': 1, '1': 2, '2': 3, length: 3 }, 1, 2, 3,"chirag" ]

// if I do name:”chirag” then it will gives me empty

Concat on non-array objects:

console.log(Array.prototype.concat.call({}, 1, 2, 3)); // [{}, 1, 2, 3]

console.log(Array.prototype.concat.call(1, 2, 3)); // [ [Number: 1], 2, 3 ]

console.log(Array.prototype.concat.call(0, 1, 2, 3)); // [ [Number: 0], 1, 2, 3 ] first no will be treated as an array another will treated like new array and merge both like above

const arrayLike = {

[Symbol.isConcatSpreadable]: true,

length: 2,

0: 1,

1: 2,

2: 99, // ignored by concat() since length is 2

};

console.log(Array.prototype.concat.call(arrayLike, 3, 4)); // [1, 2, 3, 4]

9) **filter() function –** it is used to filter the array and it will creates always creates a new array do not change the original array.

It returns those elements which passes the tests.

Filter takes 2 arguments one is callback and another is this argument.

And this callback takes three arguments which are currect element, index and array.

This filter is an iterative method. This filter calls a callback function for each element and if callback returns true value then this will be set to new array and you will get all true values into new array

const words = ["spray", "elite", "exuberant", "destruction", "present"];

const result = words.filter((word) => word.length > 6);

console.log(result) //output["exuberant", "destruction", "present"]

Question – find alll prime number in an array.

const array = [-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13];

function prime(*elem*){

    for(let i=2;i<elem;i++){

        if(elem%i === 0){

            return false

        }

    }

    return elem>1

}

console.log(array.filter(prime))// [2, 3, 5, 7, 11, 13]

Filtering invalid entries form JSON

const arr = [

  { id: 15 },

  { id: 0 },

  { id: -5 },

  { id: 0 },

  { id: 12.2 },

  {},

  { id: null },

  { id: NaN },

  { id: "undefined" },

];

let invalidcount = 0

function filterByID(*arr*){

  if(arr.id !== 0 && typeof(arr.id) === "number"){

    return true

  }

  invalidcount+=1

    return false

}

const arrByID = arr.filter(filterByID);

console.log(arrByID)// numbers will be returned which are not 0

console.log(invalidcount) //4

searching a particular query in an array see the example

const fruits = ["apple", "banana", "grapes", "mango", "orange"];

function filterItems(*arr*, *query*) {

  return arr.filter((*el*) => el.toLowerCase().includes(query.toLowerCase()));

}

console.log(filterItems(fruits, "ap")); *// ['apple', 'grapes']*

console.log(filterItems(fruits, "an")); *// ['banana', 'mango', 'orange']*

when filters used on sparse array it skips the empty slot.

**Filters on non-array objects-** this filter reads the length property of this and then access each property whose key is non negarive or less then length

const arrayLike = {

  length: 4,

  0: "a",

  1: "b",

  2: "c",

  3: "a",

};

console.log(Array.prototype.filter.call(arrayLike, (*x*) => x <= "b")) *// [ 'a', 'b', ‘c’ ]*

10) **map() function –** it is basically used to modify the array and gives new array it do not change the original array. It consists of two argument one is callback another is thisargument. And this value is used to execute when execute callback.

Return is must otherwise undefined will be displayed.

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

const map1 = array1.map((x) => x \* 2);

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

USING map to reformat new array object – means using array of object to give new modified array objects.

const array = [

{ key: 1, value: 10 },

{ key: 2, value: 20 },

{ key: 3, value: 30 },

];

const newarr = array.map(({key,value})=>{

return {[key]:value}

})

console.log(newarr[0])// [{1:10},{2:20},{3:30}]

console.log(Object.keys(newarr[0])) //[1] this is how to find keys

console.log(Object.values(newarr[0]))//[10] this is how to find val

console.log(array[0].key) // 1 if name is given

parseInt() with map() function –

let newarr = ["1", "2", "3"].map((str) => parseInt(str,10));

console.log(newarr)// here you get [1,2,3]

let newarr2 = ["1", "2", "3"].map(parseInt);

console.log(newarr2)// here you get [1,Nan,Nan]

["1.1", "2.2e2", "3e300"].map((str) => parseInt(str, 10)); // [1, 2, 3]

let newarr3 = ["1", "2", "3"].map(Number);

console.log(newarr3)// but here [1,2,3]

["1.1", "2.2e2", "3e300"].map(Number); // [1.1, 220, 3e+300]

*parseInt() here used with one argument but it takes 2 arguments first is the element and second is radix (base of number radix 10 means decimal radix 2 means binary) and in newarr2 the arguments creates problem*

Mapped array contains undefined – means when undefined or nothing is defined so it returns undefined.

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

const filterno = numbers.map((num, index) => {

if (index < 3) {

return num;

}

});

Console.log(filterno) // [1, 2, 3, undefined]

Some example of arrays of objects –

In this if you will not use return then undefined

const products = [

{ name: "sports car" },

{ name: "laptop" },

{ name: "phone" },

];

const newarr = products.map((product) => {

return product.price = 100;

});

console.log(newarr) //[100,100,100]

now using forEach in this to add price to each array object

const products = [

{ name: "sports car" },

{ name: "laptop" },

{ name: "phone" },

];

products.forEach((product) => {

return product.price = 100;

});

console.log(products)// now price will be added to each object

If I want to create a new array then use map() and simply return { ...product, price: 100 } this you will get newarr with price added to each section

Sparse array remains sparse after map() implementataion.

Using map() on non-array –

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 5, // will be ignored by map() since length is 3

};

console.log(Array.prototype.map.call(arrayLike, (x) => x \*\* 2));

// [ 4, 9, 16, 25]

11) **reduce() function –** it takes two parameters callback and initialvalue. It execute callback for each element of the array and apply certain actions by using previous values of array and another parameter is used to initialize value

Example – if I want to preform sums then callback is used and callback consist of 4 parameters

Accumulator – contains value of the previous call. On the firstcall the value will be initial value.

Currentvalue – current element of the original array. If initial value is specified then for first callback currentvalue will be array[0] if not the value will be array[1]

Currentindex – index of current element.

Array – points to the original array for each iteration.

Initialvalue – the value to which accumulator is initialized. If initial value is not specified then it will takes array 0th index value and accumulator will starts form array 1th index.

Return value will be the final result.

const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4 if not initial value then 1+2+3+4

const initialValue = 0;

const sums = array1.reduce(

(accumulator, currentValue) => accumulator + currentValue,

initialValue,

);

console.log(sums);

// Expected output: 10

If array is empty and initialvalue is not specified then error occurred.

Reducer do not accept this argument unlike other iterative methods.

It always returns a single value after accumulating each elements.

Reducer without an initial value in this example we are tracking each things accumulator, currentvalue and sum.

const array = [15, 16, 17, 18, 19];

function reducer(acc, curr, index) {

const sums = acc + curr;

console.log(

`acc: ${acc}, curr: ${curr}, index: ${index}, sums: ${sums}`,

);

return sums;

}

array.reduce(reducer);

In this example at first callback the acc = 15 because initial value is not given and curr will be 16 and sums will be 31 then another callback will be called with updated sums 31 and the sums now becomes 48 at second call and soo on.

If I give initial value then my acc will be 0 at first callback and curr will be 15 and sums will be 15 and then another callback is called and soo on.

**Sum of values in an object –** this is same as above but in object you have to pass the initialvalue with keys like

const objects = [{ x: 1 }, { x: 2 }, { x: 3 }];

const sum = objects.reduce(

(acc, curr) => acc + curr.x,

0,

);

console.log(sum); // 6

**reduce on non-array objects –** works same as above

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 99, // ignored by reduce()becauselength is 3

};

console.log(Array.prototype.reduce.call(arrayLike, (x, y) => x + y)) //9

12) **sort() function –** sort function is used to sort the array in place.

And return the reference of the same array. By default it sorts in ascending order.

If there is string given inside the array then it takes the unit place of that string convert that character into its ascii value and according to ascii value it sorts.

If there is number inside the array same for that convert to ascii value and sort according to ascci.

If the unit place of two elements are same in an array then it will checks for the second place.

const months = ["March", "Jan", "Deb", "Dec"]

months.sort()

console.log(months) // ["Deb", "Dec", "Jan", "March"] you can see Deb and Dec here it is checking according to b and c

const array1 = [10, 30, 4, 21, 100000];

array1.sort();

console.log(array1)// [10,100000,21,30,4]

sort takes one parameter which is comparefunction and it is optional.

Comparefun – it determines the order of the function. And it takes two parameters (a and b).

a – first element for comparison. And can never be undefined.

b – second element for comparison. And can never be undefined.

It should return a no:

Negative value means a should come before b.

Positive value means a should come after b.

Equal to 0 means the original format.

For hint see (a, b) => a – b this sort in ascending order.

Return value refer to the original array.

If comparefun is not been provided then undefined will be converted to string and create confusions so by using this function our undefined will be handled later.

Sorting with all the examples -

const stringArray = ["Blue", "Humpback", "Beluga"];

const numberArray = [40, 1, 5, 200];

const numericStringArray = ["80", "9", "700"];

const mixedNumericArray = ["80", "9", "700", 40, 1, 5, 200];

function compareNumbers(a, b) {

return a - b;

}

stringArray.sort(); // ['Beluga', 'Blue', 'Humpback']

numberArray.sort(); // [1, 200, 40, 5]

numberArray.sort(compareNumbers); // [1, 5, 40, 200]

numericStringArray.sort(); // ['700', '80', '9']

numericStringArray.sort(compareNumbers); // ['9', '80', '700']

mixedNumericArray.sort(); // [1, 200, 40, 5, '700', '80', '9']

mixedNumericArray.sort(compareNumbers); // [1, 5, '9', 40, '80', 200, '700']

Sorting on objects –

const arrayLike = {

length: 3,

unrelated: "foo",

0: 5,

2: 4,

};

console.log(Array.prototype.sort.call(arrayLike)// { 0: 4, 1: 5, length: 3, unrelated: "foo" }

13) join() function – join function is used to join array elements and it gives the string and this join function takes one parameter which is known as seperator

const elements = ["Fire", "Air", "Water"];

console.log(elements.join()) //output: "Fire,Air,Water"

console.log(elements.join("")) //output: "FireAirWater"

console.log(elements.join("-")) //output: "Fire-Air-Water

The parameter separator is a string if you do not give any string to join parameter then it separated by “,” by default.

If array is empty then you will get empty string in every case of separator. And if the array element is undefined or null then it will be converted to empty space.

In matrix you can control only control the upper lever separator while deeper level separator will be always use commas bydefault.

const matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9],

];

console.log(matrix.join()); // 1,2,3,4,5,6,7,8,9

console.log(matrix.join(";")); // 1,2,3;4,5,6;7,8,9 //see changes

JOIN on objects – it reads the property of this. And access those whose keys is non-negative and less then length.

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 5, // ignored by join() since length is 3

};

console.log(Array.prototype.join.call(arrayLike)) // 2,3,4

console.log(Array.prototype.join.call(arrayLike, ".")) // 2.3.4

14) with() function - This function returns the new array. And it consists of two parameters and it is basically used to replace the element of an array.

It parameters are index and value. index is used to point that index and value is used to replace that index with value.

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

console.log(arr.with(2, 6)); // [1, 2, 6, 4, 5]

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

chaining array methods.

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

console.log(arr.with(2, 6).map((x) => x \*\* 2)); // [1, 4, 36, 16, 25]

on sparse array if empty occurs then it returns undefined at that place

const arr = [1, , 3, 4, , 6];

console.log(arr.with(0, 2)); // [2, undefined, 3, 4, undefined, 6]

using this on objects-

const arrayLike = {

length: 3,

unrelated: "foo",

0: 5,

2: 4,

3: 3, // ignored by with() since length is 3

};

console.log(Array.prototype.with.call(arrayLike, 0, 1)) // [ 1, undefined, 4 ]

// undefined because after 0, 1 is not not present in obj.

16) values() function – it returns a new array iterator object that iterate over all values of an array.

const array1 = ["a", "b", "c"];

const iterator = array1.values();

for (const value of iterator) {

console.log(value);//

“a”

“b”

“c”

If I do again for of in this iterator then return undefined because it can be used once

Takes no parameter and for empty element it returns undefined

NOTE- iterable is not reusable.

If I used break statement in between then it will print till break and you again use the loop then it will start print form after the break

const arr = ["a", "b", "c", "d", "e"];

const values = arr.values();

for (const letter of values) {

console.log(letter);

if (letter === "b") {

break;

}

}

// "a" "b"

for (const letter of values) {

console.log(letter);

}

// "c" "d" "e"

Mutating during iterator:

There are no values stored in the iterator object instead this iterator points to the same reference of the array and it keeps a tract wether it is visited or not. If changes occurred in original array then the iterator value also changes.

const arr = ["a", "b", "c", "d", "e"];

const iterator = arr.values();

console.log(iterator); // Array Iterator { }

console.log(iterator.next().value); // "a"

arr[1] = "n";

console.log(iterator.next().value); // "n"

like other iterator methods this do not saves the array length at the time of creation. Which can cause infinite loop and error occurs

on sparse array it returns undefined if empty element occurs

calling values on objects:

const arrayLike = {

length: 3,

0: "a",

1: "b",

2: "c",

3: "d", // ignored by values() since length is 3

};

for (const entry of Array.prototype.values.call(arrayLike)) {

console.log(entry);

//a

//b

//c

**17) Includes() function –** this function Is used to determine whether the original array consists of particular element or not.

It returns boolean values.

const array1 = [1, 2, 3];

console.log(array1.includes(2)) // returns true

console.log(array1.includes(7) // returns false

it consists of two parameters(searchElem,fromIndex)

searchElem – the value you are searching.

fromIndex – zero based index at which search starts form.

Index is used to check whether the particular searchElem at particular index is equal to original array element if yes then true.

console.log([1, 2, 3].includes(2, 2)) // like this is false

console.log([1, 2, 3].includes(2, 1)) // this is true

If index is greater then or equal to the length of array false is returned.

But for negative index if the search elem is present in the array and negative index > array.length then it acts as noraml inclues(elem) otherwise it will check elem on index.

console.log([1, 2, 3].includes(3, -10)) // gives you true

console.log([1, 2, 3].includes(3, -100)) // gives you true

console.log([1, 2, 3].includes(3, -2)) // gives false

empty is considered as undefined if we check for undefined then if empty is present in an array then it returns true.

**Includes on objects –** it reads the length property of this and key should be non-negative.

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 1, // ignored by includes() since length is 3

};

console.log(Array.prototype.includes.call(arrayLike, 2)) // true

console.log(Array.prototype.includes.call(arrayLike, 1)) // false

18) **copyWithin() function –** It do not modifies the length of an array it basically shallow copies the particular elem address to another elem address.

It consists of three parameters target, start, end.

const array1 = ["a", "b", "c", "d", "e"];

// element at third index is copied to 0th index

console.log(array1.copyWithin(0, 3, 4)) //output ["d", "b", "c", "d", "e"]

// element at third index is copied to first index

console.log(array1.copyWithin(1, 3));

//output ["d", "d", "e", "d", "e"]

Parameters ()

Target – zero based index if not given then by default 0. If target is less then -arr.length then it by default takes 0 index. If target is greater then length then nothing is coppied.

Start – zero based index at which to start coping element from.

If start < -array.length then 0th index will be hit.

Is start>array.length then nothing will be copied.

Optional – zero based index at which we have to end coping means copy upto end but end will not be included. And it is optional too.

copyWithin() on sparse array –

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

Calling copyWithin() on objects –

const arrayLike = {

length: 5,

3: 1,

};

console.log(Array.prototype.copyWithin.call(arrayLike, 0, 3));

// { '0': 1, '3': 1, length: 5 }

console.log(Array.prototype.copyWithin.call(arrayLike, 3, 1));

// { '0': 1, length: 5 } nothing will be happened because 3 is not present within the length.

19) **toReversed() function –** same as reverse but it do not change the original array it gives us the new array with changes.

const items = [1, 2, 3];

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

const reversedItems = items.toReversed();

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

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

**20) toSorted() function –** same as sort but it do not change the original array it will gives you new array with changes.

const months = ["Mar", "Jan", "Feb", "Dec"];

const sortedMonths = months.toSorted();

console.log(sortedMonths); // ['Dec', 'Feb', 'Jan', 'Mar']

console.log(months); // ['Mar', 'Jan', 'Feb', 'Dec']

**21) every() function –** this every function is used to check whether all the elements in an array passes the test implemented by the function which is passed inside the every function as a callback.

It consists of two arguments (callback, thisargu)

const callback = (curr) => curr < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(callback)// true

Return value will be boolean true if all the elements true if not then false.

Using every on sparse array – every will not run on empty array

Console.log([2, , 2].every((elem)=>elem===2) // returns true

**every on objects –**

const arrayLike = {

length: 3,

0: "a",

1: "b",

2: "c",

3: 345, // ignored by every() since length is 3

};

console.log(

Array.prototype.every.call(arrayLike, (x) => typeof x === "string"),

); // true

If length becomes 4 then it will return false because at 3 there is no string there is number

**22) findIndex() function –** it consists of two parameters one is callback and another is thisArg. It will return the index of an array that satisfies the callback conditions if no element is satisfied then return -1 and if more then one element satisfied then return the first occurred.

const array1 = [5, 12, 8, 130, 44];

const isLargeNumber = (element) => element > 13;

console.log(array1.findIndex(isLargeNumber)) // 3

Return value will be the first elemnet which passes the test otherwise

-1.

Question – find the index of the first even number in an array

function even(elem){

if(elem%2 == 0){

return true

}

return false

}

console.log([1,3,5,6,7].findIndex(even)) // 3

sparse array empty space will be considered as undefined so we can find index of empty slot using undefined.

console.log([1, , 3].findIndex((x) => x === undefined)); // 1 like this.

findIndex() on objects –

const arrayLike = {

length: 3,

"-1": 0.1, // ignored by findIndex() since -1 < 0 key is -ve

0: 2,

1: 7.3, //

2: 4,

};

console.log(

Array.prototype.findIndex.call(arrayLike, (x) => !Number.isInteger(x)),

); // 1

**23) at() method of array –** the at method takes an integer value and according to that integer value gives us the element of the array here in this the integer value considered as index.

const array1 = [5, 12, 8, 130, 44];

let index = 2;

console.log(`An index of ${index} returns ${array1.at(index)}`);

// Expected output: "An index of 2 returns 8"

index = -2;

console.log(`An index of ${index} returns ${array1.at(index)}`);

// Expected output: "An index of -2 returns 130"

Return undefined if index will be less then array.length or greater then array.length

**at on objects –** reads the property of length.

const arrayLike = {

length: 2,

0: "a",

1: "b",

2: "c", // ignored by at() since length is 2

};

console.log(Array.prototype.at.call(arrayLike, 0)); // "a"

console.log(Array.prototype.at.call(arrayLike, 2)); // undefined

**24) forEach funtion - The forEach method runs only on array. And it runs once for each element.**

**const arr1 = ["a","b","c","d"]**

**arr1.forEach((elem)=>console.log(elem))**

**this forEach() takes two arguments one is callback and the another is thisargu. And the callback function takes three arguments one is elem which points to the currect element of the array second argument points to the index of the currect element and the third argument points to the array itself.**

**forEach always return undefined and it is not chainable. And this callback funtion is invoked only for array indexes which have values.**

**there is no way to stop or break forEach loop but we can manage by using exception.**

**const arr1 = ["a","b","c","d"]**

**const arr2 = []**

**for(let i=0;i<arr1.length;i++){**

**arr2.push(arr1[i])**

**}**

**console.log(arr2) // this is how we copy by using for loop**

**const arr1 = ["a","b","c","d"]**

**const arr2 = []**

**arr1.forEach((elem)=>arr2.push(elem)) // and this is how we do by using forEach loop**

**arr = ["a","b","c"]**

**const callback = (elem,index)=>{**

**console.log(`a[${index}] = ${elem}`)**

**}**

**arr.forEach(callback) another way to do use callback**

**// forEach function expects asynchronous callback because it do not wait for promises**

**const ratings = [5, 4, 5]**

**let sum = 0**

**const callback = async (a,b)=> a+b**

**ratings.forEach(async (elem)=>{**

**sum = await callback(sum,elem)**

**})**

**console.log(sum) // its output will be 0 because of its async behaviour. without async await the output will be 14**

**// NOW USING SECOND ARGUMENT WHICH IS THIS ARGUMENT**

**class COUNTER{**

**constructor(){**

**this.sum = 0**

**this.count = 0**

**add(arr){**

**arr.forEach(function callback(elem){**

**this.sum+=elem**

**++this.count**

**},this)**

**}**

**}**

**}**

**const obj = new Counter();**

**obj.add([2, 5, 9]);**

**console.log(obj.count);**

**console.log(obj.sum);**

**// Here this second argument which is passed is this and this is passed to the callback() after each iteration // key point - this callback should be as in tei form of funciton statement because this in array funtion binds into lexical scope //USING THIRD ARGUMENT OF CALLBACK FUNCTION WHICH POINTS TO ARRAY**

**const arr = [3, -1, 1, 4, 1, 5]**

**arr.filter((num)=> num>0)**

**arr.forEach(function callback(elem,inx,arr){**

**console.log(arr[inx-1],elem,arr[inx+1])**

**})**

**// we are now using array to find neighbour of each element in an original array**

**const arraySparse = [1, 3, , 7];**

**let numCallbackRuns = 0;**

**arraySparse.forEach((element) => {**

**console.log({ element });**

**numCallbackRuns++;**

**});**

**console.log({ numCallbackRuns }); // The callback funtion do not envoked for the missing value vo continue karr jata h when see missing**

**const arrayLike = {**

**length: 3,**

**0: 2,**

**1: 3,**

**2: 4,**

**3: 5, // forEach method reads the length property so it do not prints this index**

**};**

**Array.prototype.forEach.call(arrayLike, (x) => console.log(x));**

**// arrayLike.forEach((elem)=>{**

**// console.log(elem)**

**// }) // in objects we do not use forEach like this in this we have to use prototype which inherits items form each other**