**6. Arrays**

**1) Adding Element**:

We can add element in an array at beginning, middle or at the end position. For this purpose we have three methods.

1. push() => Add element at the end
2. unshift() => Add element at the beginning
3. splice() => remove, replace or add element on middle.

**Example**:

const number = [3, 4];

*//End-> add element at the end*

number.push(5, 6);

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

*//Beginning-> add element at the beginning*

number.unshift(1, 2);

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

*//Middle-> add element in middle*

number.splice(2, 0, 3.5);

console.log(number); *//[1, 2, 3.5, 3, 4, 5, 6];*

**splice() method**:

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements.

splice(start: number, deleteCount: number, ...items: number[])

start -> from which index we want to start

deleteCount -> how many elements we want to delete

items -> element we want to add

**2) Finding element (Primitive Type)**:

Finding element really depends what we are works with primitive type or reference type in an array. In primitive type for find element in an array we can use the following method.

**a) indexOf()**:

The indexOf() method searches the array for the specified item, and returns its position (index).

The search will start at the specified position, or at the beginning if no start position is specified, and end the search at the end of the array.

Returns -1 if the item is not found.

If the item is present more than once, the indexOf method returns the position of the first occurrence.

Note: The first item has position 0, the second item has position 1, and so on.

**Example**:

const number = [2, 3, 4, 5, 6, 7, 8, 9, 1, 2];

*//find the index of element 3 => 1*

console.log(number.indexOf(3)); *//1*

*//return -1 if element not present in array*

console.log(number.indexOf(11)); *//-1*

*//start find from index 2 and find index of element 7 => 5*

console.log(number.indexOf(7, 2)); *//5*

*//if more than one element return the index of first element*

console.log(number.indexOf(2)); *//0*

**b) lastIndexOf()**:

The lastIndexOf() method searches the array for the specified item, and returns its position.

The search will start at the specified position or at the end if no start position is specified, and end the search at the beginning of the array.

Returns -1 if the item is not found.

If the item to search for is present more than once, the lastIndexOf method returns the position of the last occurrence.

lastIndexOf(searchElement: number, fromIndex?: number): number

Returns the index of the last occurrence of a specified value in an array.

searchElement — The value to locate in the array.

fromIndex — The array index at which to begin the search. If fromIndex is omitted, the search starts at the last index in the array.

**Example**:

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

console.log(numbers.lastIndexOf(1)); *//5*

console.log(numbers.lastIndexOf(1, 4)); *//0*

**c) includes()**:

The includes() method determines whether an array contains a specified element. This method returns true if the array contains the element, and false if not.

**Note**:

The includes() method is case sensitive.

**Example**:

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

console.log(numbers.includes(5)); *//true*

console.log(numbers.includes(8)); *//false*

*//serch element -> 2, start serch from third index*

console.log(numbers.includes(2, 2)); *//false*

*array*.includes(element, start)

element => Required. The element to search for

start => Optional. Default 0. At which position in the array to start the search

**3) Finding element (Reference Type)**:

If the array is reference type then we use the following method to find element in an array.

**a) find()**:

The find() method returns the value of the first element in the array that satisfies the provided testing function. Otherwise undefined is returned.

The find method executes the callback function once for each index of the array until it finds one where callback returns a true value. If such an element is found, find immediately returns the value of that element. Otherwise, find returns undefined. callback is invoked for every index of the array from 0 to length - 1 and is invoked for all indexes, not just those that have been assigned values. This may mean that it's less efficient for sparse arrays than other methods that only visit indexes that have been assigned a value.

**Example**:

const courses = [

{ id: 1, name: "a" },

{ id: 2, name: "b" },

{ id: 3, name: "c" }

];

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

return course.name === "a";

});

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

const course1 = courses.find(course => course.name == "b");

console.log(course1); *//{id: 2, name: "b"}*

**b) findIndex()**:

The findIndex() method returns the index of the first element in the array that satisfies the provided testing function. Otherwise, it returns -1, indicating no element passed the test.

The findIndex method executes the callback function once for every array index 0..length-1 (inclusive) in the array until it finds one where callback returns a truthy value (a value that coerces to true).

If such an element is found, findIndex immediately returns that found element's index. If the callback never returns a truthy value or the array's length is 0, findIndex returns -1. Unlike some other array methods such as Array.some, in sparse arrays the callback is called even for indexes of entries not present in the array.

**Example**:

const courses = [

{ id: 1, name: "a" },

{ id: 2, name: "b" },

{ id: 3, name: "c" }

];

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

return course.name === "a";

});

console.log(course); *//0*

const course1 = courses.findIndex(course => course.name === "d");

console.log(course1); *//-1*

**4) Arrow Function**:

Arrow functions (also called “fat arrow functions”) are undoubtedly one of the more popular features of ES6. They introduced a new way of writing concise functions.

**Normal Function**:

function timesTwo(params) {

return params \* 2;

}

console.log(timesTwo(4)); *// 8*

const courses = [

{ id: 1, name: "a" },

{ id: 2, name: "b" },

{ id: 3, name: "c" }

];

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

return course.name === "c";

});

console.log(course); *//2*

**Arrow function**:

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

return course.name === "c";

});

console.log(course); *//2*

var timesTwo = params => params \* 2;

console.log(timesTwo(4)); *//8*

const c1 = courses.findIndex(course => course.name == "c");

console.log(c1); *//2*

**5) Removing Element**:

For removing and element from an array we can the following three elements. By using this method we can remove element of an array from beginning, middle or end position.

1. pop() => remove an element from last
2. shift() => remove an element from beginning
3. splice() => remove an element from middle

**a) pop()**:

Removes the last element from an array and returns it.

**Example**:

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

const last = numbers.pop();

console.log(last); *//4*

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

**b) shift()**:

Removes the first element from an array and returns it.

**Example**:

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

const first = numbers.shift();

console.log(first); *//1*

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

**c) splice()**:

Removes elements from an array and, if necessary, inserts new elements in their place, returning the deleted elements.

splice(start: number, deleteCount?: number): number[] (+1 overload)

start => The zero-based location in the array from which to start removing elements.

deleteCount => The number of elements to remove.

**Example**:

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

const removeNumber = numbers.splice(2, 1);

console.log(removeNumber); *//[3]*

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

**Note**:

If we pass a single parameter to splice() method then the method receive it as index number and remove the element from index to all the rest element and return it as an array.

**Example**:

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

const num = numbers.splice(0); *//[ 1, 2, 3, 4 ]*

const num = numbers.splice(1); *//[ 2, 3, 4 ]*

const num = numbers.splice(2); *//[ 3, 4 ]*

const num = numbers.splice(3); *//[ 4 ]*

**Note**:

The splice() method changes the contents of an array by removing or replacing existing elements and/or adding new elements.

**6) Emptying an array**:

Emptying means remove all the elements from an array. There are a few difference solutions for this.

**Solution-1**:

Reassign an array.

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

numbers = [];

console.log(numbers); *//[]*

Here the array [1, 2, 3, 4] still in the memory, so if there are no other references to this object, it will

removed by the garbage collector. If the array numbers have any other reference then [1, 2, 3, 4] is not removed by garbage collector. When we reassign numbers = [] numbers it will pointing a new object.

**Example**:

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

let another = numbers;

numbers = [];

console.log(numbers); *//[]*

console.log(another); *//[1, 2, 3, 4]*

So if our array has multiple references we cannot use this we have to use another solution.

**Solution-2**:

Set the lengths property to zero and this will truncate the array. It will remove all elements.

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

let another = numbers;

numbers.length = 0;

console.log(numbers); *//[]*

console.log(another); *//[]*

**Solution-3**:

Use the splice() method. By using we can be emptying an array. The other array which uses the array reference is also empty.

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

let another = numbers;

numbers.splice(0, numbers.length); *//[1, 2, 3, 4]*

*//or numbers.splice(0); //[1, 2, 3, 4]*

console.log(numbers); *//[]*

console.log(another); *//[]*

**Solution-4**:

Use pop() method. pop() method removes the last element in the array. Now we can keep this in a loop, and keep calling this method, as long as we have an element in this array.

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

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

let another = numbers;

while(numbers.length > 0){

numbers.pop();

}

console.log(numbers); *//[]*

console.log(another); *//[]*

This approach is not recommended because if we dealing with a large array, if we have a million object in that array, there’s going to be a performance cost. We have to calling this pop method a million times.

**Note**:

From the above four approach the 2nd approach is recommended “Set the lengths property to zero”. If we have a single reference array the first solution is perfect.

**7) Combining and Slicing array**:

**Combining array (concat())**:

In JavaScript Array objects actually have a method called concat() that allows us to take an array, combine it with another array, and return a new array.

*array1*.concat(array2, array3, ..., arrayX)

array2, array3, ..., arrayX => Required. The arrays to be joined

**Example**:

let first = [1, 2, 3];

let second = [4, 5, 6];

const combined = first.concat(second);

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

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]*

**Example**:

In reference type

const first = [{ id: 1 }];

const second = [4, 5, 6];

*//here references will be coped*

const combined = first.concat(second);

console.log(combined); *//[{…}, 4, 5, 6] here id: 1*

**Example**:

const first = [{ id: 1 }];

const second = [4, 5, 6];

*//here references will be coped*

const combined = first.concat(second);

first[0].id = 20;

console.log(combined); *//[{…}, 4, 5, 6] here id: 20*

**Slicing an array (slice())**:

The slice() method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). The original array will not be modified.

**Example**:

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

const slice = numbers.slice(2, 4);

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

const slice1 = numbers.slice(2);

console.log(slice1); *//[3, 4, 5, 6]*

**Note**:

In primitive type value will be copied in target array but if we have object in our array object will not coped, the references are coped.

**8) The Spread Operator**:

In ES6 we can use Spread Operator for combined, copy array and many more.

**Example**:

const first = [1, 2, 3];

const second = [4, 5, 6];

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

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

const custom = [...first, 'a', ...second, 'b'];

console.log(custom); *//[1, 2, 3, "a", 4, 5, 6, "b"]*

const copy = [...combined];

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

**9) Iterating an Array**:

We can iterate an array in JavaScript by using the following method.

**Example**:

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

*//using for-of loop*

for (let number of numbers) {

console.log(number); *//1 2 3 4 5 6*

}

*//using forEach loop*

numbers.forEach(function(number) {

console.log(number); *//1 2 3 4 5 6*

});

*//using arrow function*

numbers.forEach(number => console.log(number)); *//1 2 3 4 5 6*

**10) Joining array (join())**:

The join() method creates and returns a new string by concatenating all of the elements in an array (or an array-like object), separated by commas or a specified separator string.

**Example**:

const numbers = [1, 2, 3];

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

console.log(join); *//1,2,3*

const message = "This is my first message";

const parts = message.split(" ");

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

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

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

**11) Sorting arrays**:

**sort() method**:

We can use sort() method to short an array in JavaScript. The sort() method sorts an array alphabetically.

The name of the function used to determine the order of the elements. If omitted, the elements are sorted in ascending, ASCII character order.

**Example**:

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

const sort = numbers.sort();

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

**reverse() method**:

The reverse() method reverses the elements in an array. We can use it to sort an array in descending order.

**Example**:

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

const sort = numbers.sort(); *//[1, 2, 3, 4, 5, 6]*

const reverse = numbers.reverse();

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

**Object type**:

const courses = [

{ id: 1, name: "Node.js" },

{ id: 2, name: "JavaScript" },

{ id: 3, name: "Electron JS" }

];

courses.sort(function(a, b) {

if (a.name > b.name) return 1;

if (a.name < b.name) return -1;

return 0;

});

console.log(courses);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {id: 3, name: "Electron JS"}*

*1: {id: 2, name: "JavaScript"}*

*2: {id: 1, name: "Node.js"}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

**Note**:

In object type sort() function works based on ASCII code. So, the result varied for uppercase and lowercase character. To solve this problem, we have to convert all letter in uppercase or lowercase.

**Problem**:

const courses = [

{ id: 1, name: "Node.js" },

{ id: 2, name: "javaScript" },

{ id: 3, name: "Electron JS" }

];

courses.sort(function(a, b) {

if (a.name < b.name) return -1;

if (a.name > b.name) return 1;

return 0;

});

console.log(courses);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {id: 3, name: "Electron JS"}*

*1: {id: 1, name: "Node.js"}*

*2: {id: 2, name: "javaScript"}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

**Solution**:

const courses = [

{ id: 1, name: "Node.js" },

{ id: 2, name: "javaScript" },

{ id: 3, name: "Electron JS" }

];

courses.sort(function(a, b) {

nameA = a.name.toUpperCase();

nameB = b.name.toUpperCase();

if (nameA > nameB) return 1;

if (nameA < nameB) return -1;

return 0;

});

console.log(courses);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {id: 3, name: "Electron JS"}*

*1: {id: 1, name: "Node.js"}*

*2: {id: 2, name: "javaScript"}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

**12) Testing the Elements of an array**:

We have two new method in modern JavaScript, every() and some(). By using this method we can testing the elements of an array.

**every() method**:

The every() return true if all value is positive and false otherwise

**Example**:

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

const allPositive = numbers.every(function(number){

return number >= 0;

});

console.log(allPositive); *//true*

const numbers1 = [1, -2, 3, 4, 5];

*//or arrow function*

console.log(numbers1.every(number => number >= 0)); *//false*

**some() method**:

The some() return true if at least one value is positive.

**Example**:

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

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

return value >= 0;

});

console.log(atLeastOnePositive); *//true*

*//arrow function*

const numbers1 = [-1, -2, -3, -4];

console.log(numbers1.some(number => number >= 0)); *//false*

**Note**:

The every() and some() method are new in ES6 some old browser may not support this.

**13) Filtering an Array**:

We use filter() method for filtering an array. The filter() method creates an array filled with all array elements that pass a test (provided as a function). filter() does not execute the function for array elements without values. filter() does not change the original array.

**Example**:

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

const filter = numbers.filter(function(value){

return value >= 0;

});

console.log(filter); *//[1, 2, 5, 6]*

*//using flatArrow function*

const filter1 = numbers.filter(value => value >= 0);

console.log(filter1); *//[1, 2, 5, 6]*

**14) Mapping an Array**:

The map() method is one of the powerful method in modern JS. By using this we can map each element of an array with something else.

map() and filter() method returns an array.

**Example**:

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

const filtered = numbers.filter(n => n >= 0);

const items = filtered.map(n => '<li>' + n + '</li>' );

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

*//join the element and create a string*

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

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

**Example**:

map() with object

*//map with object*

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

const filtered = numbers.filter(n => n >= 0);

const items = filtered.map(n => {

const obj = { value: n };

return obj;

});

console.log(items);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {value: 1}*

*1: {value: 2}*

*2: {value: 3}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

**Example**:

Using flat arrow function.

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

const filtered = numbers.filter(n => n >= 0);

const items = filtered.map(n => { value: n});

console.log(items); *//[undefined, undefined, undefined]*

*/\**

*we will get undefined because {} in an arrow function represent a code block. When the JavaScript engin try to pass this arrow function it think { value: n } as a code block. If we want to returning an object we should put that object in ({ value: n }).*

*\*/*

If we run the above program, we will get undefined because {} in an arrow function represent a code block. When the JavaScript engine try to pass this arrow function it think { value: n } as a code block. If we want to returning an object we should put that object in ({ value: n }).

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

const filtered = numbers.filter(n => n >= 0);

const items = filtered.map(n => ({ value: n }));

console.log(items);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {value: 1}*

*1: {value: 2}*

*2: {value: 3}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

Now map() and filter() are chainable. We can call them one after another in a chain.

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

const number = numbers.

filter(n => n >= 0).

map(n => ({value : n}));

console.log(number);

*/\**

*(3) [{…}, {…}, {…}]*

*0: {value: 1}*

*1: {value: 2}*

*2: {value: 3}*

*length: 3*

*\_\_proto\_\_: Array(0)*

*\*/*

This above program return an array and we can call filter() method to filter the array.

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

const number = numbers

.filter(n => n >= 0)

.map(n => ({ value: n }))

.filter(obj => obj.value >= 2);

console.log(number);

*/\**

*(2) [{…}, {…}]*

*0: {value: 2}*

*1: {value: 3}*

*length: 2*

*\_\_proto\_\_: Array(0)*

*\*/*

**15) Reducing an Array**:

We can use reduce() function for reducing an array. The reduce() method executes a reducer function (that you provide) on each member of the array resulting in a single output value.

**Example**:

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

*//initially a(accumulator) = 0 and c(cuttentValue) = 1 (ist element)*

*//a=0; c(1st element)=1 => a=1;*

*//a=1, c(2nd element)=-1 => a=0*

*//a=0, c(3rd element)=2, a=2*

*//a=2, c(4th element)=3, a=5*

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

return accumulator + currentValue;

}, 0);

console.log(sum); *//5*

*//or arrow function*

const sum1 = numbers.reduce((accumulator, currentValue) => accumulator + currentValue);

console.log(sum1); *//5*

6. Arrays