**39.Explain JS Array Methods.**

* The JavaScript method toString() converts an array to a string of (comma separated) array values.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.toString();

* The join() method also joins all array elements into a string. It behaves just like toString(), but in addition you can specify the separator

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.join(" \* ");

* The pop() method removes the last element from an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.pop();

* But here pop() method returns the value that was "removed out":

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
document.getElementById("demo").innerHTML = fruits.pop();

* The push() method adds a new element to an array (at the end):

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");

* But Result changed in this case. Result will be the length of an array after adding “Kiwi”.

document.get..Id("demo").innerHTML = fruits.push(" kiwi ");

* The shift() method removes the first array element and "shifts" all other elements to a lower index.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();

* But here shift() method returns the value that was "shifted out":

document.get..Id("demo").innerHTML = fruits.shift();

* The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

* But here the unshift() method returns the new array length.

document.get..Id("demo").innerHTML = fruits.shift();

* Array elements can be deleted using the JavaScript operator delete. Using delete leaves undefined holes in the array.

Use pop() or shift() instead.

* The concat() method creates a new array by merging (concatenating) existing arrays. The concat() method can take any number of array arguments:

const myGirls = ["Cecilie", "Lone"];  
const myBoys = ["Emil", "Tobias", "Linus"];

const myKids = ["Tuna", "suna", "jina"];  
  
const myChildren = myGirls.concat(myBoys,myKids);

* The concat() method can also take strings as arguments:

const arr1 = ["Emil", "Tobias", "Linus"];  
const myChildren = arr1.concat("Peter");

* The splice() method can be used to add new items to an array. The first parameter (2) defines the position **where** new elements should be **added** (spliced in).The second parameter (0) defines **how many** elements should be **removed**.The rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

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

* The splice() method returns an array with the deleted items:

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

* We can use splice() to remove elements without leaving "holes" in the array. The first parameter (0) defines the position where new elements should be added (spliced in).The second parameter (1) defines how many elements should be removed.The rest of the parameters are omitted.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(0, 1);

* The slice() method slices out a piece of an array into a new array.

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

* The slice() method can take two arguments like slice(1, 3).
* The sort() method sorts an array alphabetically.

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

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

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

* By default, the sort() function sorts values as **strings**. But for numbers we use **compare function**. The purpose of the compare function is to define an alternative sort order. The compare function should return a negative, zero, or positive value, depending on the arguments. After you have sorted an array, you can use the index to obtain the highest and lowest values.

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

* You can use Math.max.apply to find the highest number in an array. Math.max.apply(null, [1, 2, 3]) is equivalent to Math.max(1, 2, 3).

function myArrayMax(arr) {  
  return Math.max.apply(null, arr);  
}

* You can use Math.min.apply to find the lowest number in an array:

function myArrayMin(arr) {  
  return Math.min.apply(null, arr);  
}

* const cars = [  
    {type:"Volvo", year:2016},  
    {type:"Saab", year:2001},  
    {type:"BMW", year:2010}  
  ];

cars.sort(function(a,b){return a.year - b.year});//sort obj array

* The filter() method creates a new array with array elements that passes a test.

const numbers = [45, 4, 9, 16, 25];  
const over18 = numbers.filter(myFunction);  
  
function myFunction(value, index, array) {  
  return value > 18;  
}

* The reduce() method runs a function on each array element to produce (reduce it to) a single value.The reduce() method works from left-to-right in the array. See also reduceRight().

const numbers = [45, 4, 9, 16, 25];  
let sum = numbers.reduce(myFunction);  
  
function myFunction(total, value, index, array) {  
  return total + value;  
}

* The reduce() method can accept an initial value:

const numbers = [45, 4, 9, 16, 25];  
let sum = numbers.reduce(myFunction, 100);  
  
function myFunction(total, value) {  
  return total + value;  
}

* The reduceRight() method runs a function on each array element to produce (reduce it to) a single value.The reduceRight() works from right-to-left in the array.
* The every() method check if all array values pass a test.

const numbers = [45, 4, 9, 16, 25];  
let allOver18 = numbers.every(myFunction);  
  
function myFunction(value, index, array) {  
  return value > 18;  
}

* The some() method check if some array values pass a test.

const numbers = [45, 4, 9, 16, 25];  
let someOver18 = numbers.some(myFunction);  
  
function myFunction(value, index, array) {  
  return value > 18;  
}

* The indexOf() method searches an array for an element value and returns its position. Array.indexOf() returns -1 if the item is not found.

If the item is present more than once, it returns the position of the first occurrence. *array*.indexOf(*item*,*start*). item Required. The item to search for. start Optional. Where to start the search. Negative values will start at the given position counting from the end, and search to the end.

const fruits = ["Apple", "Orange", "Apple", "Mango"];  
let position = fruits.indexOf("Apple") + 1;

* Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.
* The find() method returns the value of the first array element that passes a test function.

const numbers = [4, 9, 16, 25, 29];  
let first = numbers.find(myFunction);  
  
function myFunction(value, index, array) {  
  return value > 18;  
}

* The findIndex() method returns the index of the first array element that passes a test function.

const numbers = [4, 9, 16, 25, 29];  
let first = numbers.findIndex(myFunction);  
  
function myFunction(value, index, array) {  
  return value > 18;  
}

* The Array.from() method returns an Array object from any object with a length property or any iterable object. Array.from("ABCDEFG");
* The Array.keys() method returns an Array Iterator object with the keys(index) of an array.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
const keys = fruits.keys();  
  
for (let x of keys) {  
  text += x + "<br>";  
}

* Includes allows us to check if an element is present in an array (including NaN, unlike indexOf).

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
  
fruits.includes("Mango");

Differences between JavaScript Objects and Maps:

|  |  |  |
| --- | --- | --- |
|  | **Object** | **Map** |
| **Iterable** | Not directly iterable | Directly iterable |
| **Size** | Do not have a size property | Have a size property |
| **Key Types** | Keys must be Strings (or Symbols) | Keys can be any datatype |
| **Key Order** | Keys are not well ordered | Keys are ordered by insertion |
| **Defaults** | Have default keys | Do not have default keys |