JavaScript Array Methods (functions)

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>

<https://www.javascripttutorial.net/javascript-array-methods/>

<https://vegibit.com/most-useful-javascript-array-functions/>

[**https://2ality.com/2022/05/processing-arrays-non-destructively.html**](https://2ality.com/2022/05/processing-arrays-non-destructively.html)

* I’m going to talk about some JavaScript Array methods or functions.
  + First off, just a little bit about JavaScript Arrays
  + The JavaScript Array is an object which allows us to store a list of items.
  + JavaScript Arrays are bound by certain characteristics:
    - They are resizable.
    - They can contain various data types, like Boolean, string, int, & float, even other objects, functions, or arrays.
    - They are zero-indexed, so their index starts at 0 and progresses incrementally, the last index can be referenced with a -1.
* There are many ways to create and populate an array object:

let pets = [“dog”, “cat”, “rabbit”, “owl”];

let pets = new Array(“cat”, “rabbit”);

pets.push(“owl”); # push() method adds item to end of array

pets.unshift(“dog”); # unshif() method adds item to beginning

console.log(pets); # displays [“dog”, “cat”, “rabbit”, “owl”]

* …and the Array object has different methods for manipulating arrays in different ways:
  + Adding/removing items: push(), unshift(), pop(), shift(), splice(), and slice().
  + Selecting items: indexOf(), includes(), find(), and findIndex().
  + High-order methods: map(), filter(), reduce(), every(), some(), sort(), forEach()
  + Merging Arrays: concat()
  + Making Arrays: of(), from()
  + Flattening Arrays: flat(), flatMap()
* These are just a few, there are many more. We will be focusing mostly on the high-order methods.
* I just want to talk real quick about the indexOf() method. I found this use case for it, and I thought it was kind of cool so this will be our warm-up before we get to the good stuff.
  + Demo
* Now let’s move on to some of the high-order methods. First, some info about these methods:
  + They all take an array as an input to return a result.
  + Some return a new array with updated or requested changes. Others may just iterate over the array.
  + They are all non-destructive. NO changes are made to the original array.
* Array.map()
  + The array.map() method is used to change the elements in an array in a way specified by the user, and returns a new array with the updated elements.
  + Syntax: arrayObject.map(callback[,contextObject]);
  + Map takes 2 arguments, only the first is required, it can be an expression or a function that returns a value, which is added to the new array.
  + Callback Syntax: function callback(currentElement, index, array){*...*}
    - currentElement: element that is passed in from the array
    - index: the current index of the currentElement
    - array: the original array that is passed through map
  + Don’t use map if:
    - Callback function doesn’t return a value
    - Don’t need to use the return array
  + Use forEach() instead
* Array.filter()
  + Takes an array, returns a filtered array, based on a condition
  + Syntax: arrayObject.filter(callback, contextObject);
  + Similar to map, filter takes two arguments where only the first is a required callback function.
  + Callback Syntax: function callback(currentElement, index, array){*...*}
    - currentElement: element that is passed in from the array
    - index: the current index of the currentElement
    - array: the original array that is passed through filter
  + If you want to filter the array and make changes to it at the same time, map() may be a better option.
* Array.reduce()
  + Reduce takes an array and reduces it to a single value after performing a requested operation.
  + Reduce has 2 argument, the callback and an optional initialValue
  + Syntax: array.reduce(callbackFn [, initialValue])
  + Callback Syntax: function callbackFn(previousValue, currentValue, currentIndex, array) *{ /\*\*/ }*
    - initialValue: the value that get iterated over, if supplied
    - previousValue: value at array[0] by default, but can be supplied by user
    - currentValue: value at array[0] if user supplies initialValue, otherwise value at array[1]
    - array: the original array that will be reduced
  + Things to be aware of:
    - Any updates made to the array after iteration begins will not be counted by reduce (i.e.: append).
    - Items deleted from array before their turn in the sequence will not be counted either.
  + Array.reduceRight() acts like reduce, but starts at the last element in the array and works backward.
* Array.every()
  + The every() method is used to check to see if all elements in an array meet a certain condition. If they do, it returns true, if not it returns false.
  + Every() does not change the original array and it does not return an updated array.
  + Syntax: arrayObject.every(callback[, thisArg])
  + Callback Syntax: function callback(currentElement, index, array){*//...*}
  + Like other Array methods every takes a mandatory callback function and the optional thisArg, which allows thisArg to reference itself from inside the callback function with the this value.
  + The syntax of the callback is similar to map and filter, accepting three arguments, the current element, the index, and the array itself.
  + When called every() will run until it returns a false (falsy) value or reaches the end of the array.
* Array.some()
  + Some() is similar to every; however, it checks to see if at least one item in the array meets the condition, if so, it returns true.
  + Syntax: arrayObject.some(callback[, thisArg]);
  + Callback Syntax: function callback(currentElement [[, currentIndex], array]){ *// ...}*
  + When called some() will run until it returns a true (truthy) value or reaches the end of the array.
  + If some() is called on an empty array it will always return false.
  + Some() determines the range of elements to be checked before running so it will not check any elements that are added after it is invoked.
* Array.sort()
  + Sort() takes the elements of an array and sorts them in ascending order based on UTF-16 code values.
  + Sort() takes an optional compare function.
  + Unlike the other methods we’ve covered sort() does not make a new sorted array, instead it sorts and updates the current array.
  + Syntax: array.sort(comparefunction)
  + Compare() Syntax: function compare(a,b) {*// ...* }
  + Essentially, if the compare function is not provided, all values are converted to strings and sorted by string comparison. So, [‘b’, ‘e’, ‘d’, ‘c’, ‘a’] becomes [‘a’, ‘b’, ‘c’, ‘d’, ‘e’] and [3, 5, 1, 30, 11] becomes [“1”, “11”, “3”, “30”, “5”].
  + The compare() function takes two arguments, sort order is determined by what compare returns:
    - If compare(a, b) < 0; a is listed before b
    - If compare(a, b) > 0; b is listed before a
    - If compare(a, b) === 0; the order remains unchanged
  + If you have to sort words with non-ASCII special characters like é, è, etc., you can use the localeCompare() function.
* Array.forEach()
  + The forEach() method is used to run a function on each element of an array.
  + Unlike map() and reduce(), forEach() cannot be chained and always returns undefined.
  + forEach() cannot be used with break, continue, or asynchronous commands like promises and async/await.
  + Syntax: Array.forEach(callback [, thisArg]);
  + Callback Syntax: function callback(currentElement, index, array){*...*}
    - currentElement: element that is passed in from the array
    - index: the current index of the currentElement
    - array: the original array that is passed through forEach
  + forEach() does not modify the array on which it is called, but the array can be altered by the callback function.