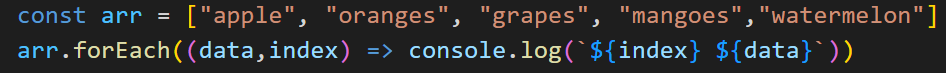
ARRAY METHODS

**forEach**:

* *arr.forEach((item, index)=>{....callbackfn logic})*
* used to simply loop over each array item and perform some action
* original array untouched
* returns nothing  
  

**map**:

* *arr.map((item,index)=> {....callbackfn logic})*
* map is the concept of transforming a set into another type of set, while preserving structure( which means Array.map returns an array of the same length every single time)
* used to modify each array item, thereby creating a new modified array out of the old one
* original array untouched
* returns a new array  
  A computer code with colorful text

  Description automatically generated with medium confidence

**filter**:

* *arr.filter((item,index)=> {....callbackfn logic})*
* used to filter out array items based on some condition
* in callback fn, we write condition to check if array item filtered out or remains
* if callback fn returns true, array item passed to resulting array else filtered out
* original array untouched
* returns a new array (contains filtered pass items)  
  A screen shot of a computer screen

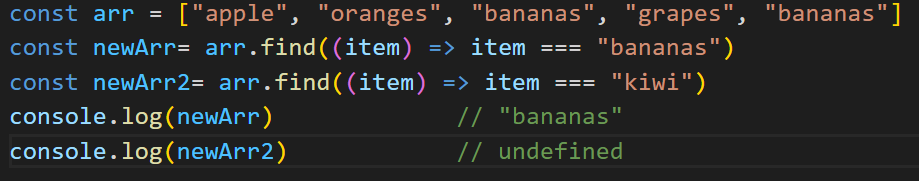
  Description automatically generated

**concat**:

* *mainArray.concat(value1,value2,arr1,arr2,....)*
* used to concatenate multiple arrays/values to mainArray
* original array untouched
* returns a new array (concatenated)  
  A black screen with colorful text

  Description automatically generated

**find**:

* *arr.find((item)=> {....callbackfn logic})*
* used to find the “first” array item that satisfies certain condition
* iterates over array items and stops immediately if callback function returns true
* callback function: if it returns true, execution stops, and that array element is returned as a result
* original array untouched
* returns an array item (any data-type) if callback function returns true. If none satisfy criteria, returns undefined.  
  

**findIndex**:

* *arr.findIndex((item)=> {....callbackfn logic})*
* used to find the index of “first” array item that satisfies certain condition
* same as arr.find() method, difference is that it returns index of matched array element
* used mostly when array values are non-primitives (can be used with primitives as well, though, but indexOf is more preferrable)
* returns index of array item if callback function returns true. If none satisfy criteria, returns -1
* original array untouched  
  A screen shot of a computer code

  Description automatically generated

**indexOf**:

* *arr.indexOf(checkValue)*
* used to verify if checkvalue exists in array and, if yes, return it’s index (first match only)
* if checkValue doesn’t exist in array, returns -1.
* very similar to arr.findIndex(), difference is that it works only for primitive array values
* won’t work for non-primitive array values
* no callbackfn here, need to enter whole value
* original array untouched  
  A screen shot of a computer program

  Description automatically generated

**lastIndexOf**:

* everything same as indexOf, only difference is that search starts from last index and ends at first index.  
  A screen shot of a computer code

  Description automatically generated

**includes**:

* *arr.includes(checkValue)*
* similar to find() and indexOf() . Only difference is it doesn’t return matched array entry or index, rather it returns a Boolean.
* checkValue should be of primitive data type only
* used to verify if checkvalue exists in array. If yes, returns true, else false.
* original array untouched  
  A screenshot of a computer screen

  Description automatically generated

**some**:

* *arr.some((data,index)=> {callbackfn logic...})*
* used to check if callbackfn condition is satisfied for at least one array item
* if callback condition is satisfied at least once, result is true, else false.
* original array untouched  
  

**every**:

* *arr.every((data,index)=> {callbackfn logic...})*
* used to check if callbackfn condition is satisfied for all array items
* if callback condition is satisfied for all array items, result is true, else false.
* original array untouched  
  A computer screen shot of text

  Description automatically generated

**push**:

* *arr.push(value1,value2,...)*
* used to add one or more elements into an array towards the end
* to push an "array" of values make sure to spread them first
* returns new length of array
* modifies original array  
  A black screen with green text

  Description automatically generated

**pop**:

* *arr.pop()*
* removes last element of array
* no parameters passed
* returns array element removed
* modifies original array  
  A black background with text

  Description automatically generated

**unshift**:

* *arr.unshift(value1,value2,...)*
* used to add elements into an array from 0th index(start)
* to push an "array" of values make sure to spread them first
* returns new length of array
* modifies original array  
  A black screen with green text

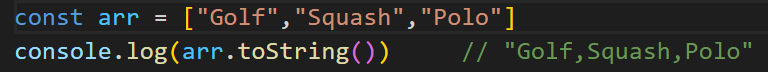
  Description automatically generated

**shift**:

* *arr.shift()*
* used to remove first element of array
* no parameters passed
* returns array element removed
* modifies original array  
  A black background with white text

  Description automatically generated

**toString**:

* *arr.toString()*
* used to convert an array to a string
* all array items joined with comma separator
* returns a string
* original array untouched  
  

**join**:

* arr.join(seperator)
* used to convert an array into a string by specifying a separator
* similar to a toString() but allows us to specify a separator
* all array items joined with separator ( by default separator is comma)
* returns a string
* original array untouched  
  A black screen with text and colorful smileys

  Description automatically generated with medium confidence

**fill**:

* *arr.fill(constValue, startIndex:optional,endIndex:optional)*
* used to replace a range of array items with a constant value
* [startindex and endindex)
* startindex and endindex can be used to specify where to start and end the replace process
* if endindex is not specified, it is equal to (length+1)
* modifies original array  
  A computer screen with text

  Description automatically generated

**slice**:

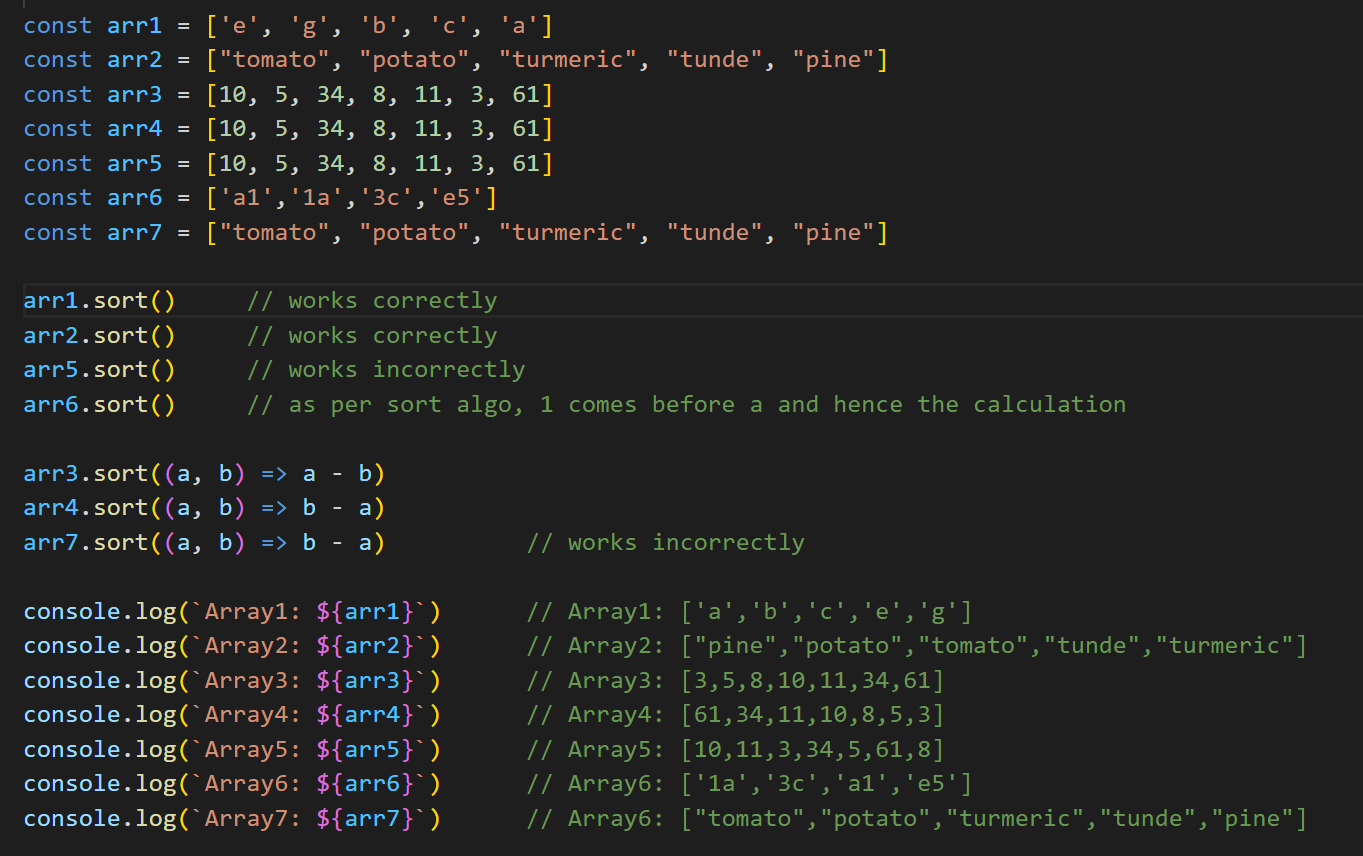
* *arr.slice(startindex:optional,endindex:optional)*
* used to slice part of an array and store it in a new array
* if startindexindex and endindex not specified, whole array returned
* if endindex is not specified, it is equal to (length+1)
* cutting occurs - [startindex, endindex)
* returns a new sliced array
* original array untouched  
  A screen shot of a computer

  Description automatically generated

**splice**:

* *arr.splice(startIndex:op,removeCount:op,addValue1:op,addValue2:op,...)*
* used to perform two kind of operations:  
   - remove existing elements  
   - remove existing elements AND add new elements
* *startIndex:* from where we want to start operation
* *removeCount:* remove elements from index = [startIndex,startIndex+removecount]
* *addValue(s):* add values to array at index = startIndex (after remove operation is done)
* if only startindex present, it removes elements from array
* if startindex and removeCount present, it removes elements from array
* if startindex and removeCount and addValues(s) present, it removes and adds elements from/to array
* returns an empty array if no params defined ( and original array untouched)
* - returns an array of removed items if params defined (and modifies original array)  
  

**sort**:

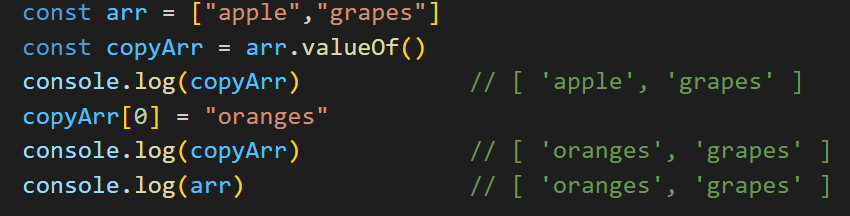
* *Type1: arr.sort()*
* *Type 2: arr.sort((a,b)=>{callbackfn logic…})*
* used to sort an array of values (strings,objects,numbers etc)
* based on situation we use either type1 or type2 to sort an array of values
* sorting algorithm uses certain calculation to compare two values (type1)
* for type2, we provide our custom algorithm
* *Type1: arr.sort()*  
   - can only perform an ascending sort, no descending sort  
   - use it to sort an array of CHARACTERS (a comes first and z comes last)  
   - use it to sort an array of strings  
   - don’t use it to sort an array of numbers  
   - cannot be used to sort an array of objects
* *Type 2: arr.sort((a,b)=>{callbackfn logic…})*  
   - use it to sort an array of numbers  
   - use it to sort an array of objects  
   - don’t use to sort an array of strings/characters
* How sort callbackfn works?  
   - exact sorting algorithm I don’t know  
   - if callbackfn returns value:  
   < 0 : a comes first, b comes second  
   > 0 : b comes first, a comes second  
   === 0 : no order change  
   - ascending sort : Array.sort((a,b) => a - b)  
   - descending sort : Array.sort((a,b) => b - a)
* returns a sorted array
* modifies original array  
  

**reverse**:

* *arr.reverse()*
* used to reverse order of array
* can be used on any kind of array
* returns reversed array
* original array modified  
  A computer screen with text and numbers

  Description automatically generated

**valueOf**:

* *arr.valueOf()*
* used to create a shallow copy of array
* returns copied array
* original array untouched (but sice it is a shallow copy, hence,a change in properties of copy array leads to change in original array)  
  

**isArray**:

* *Array.isArray(arr)*
* used to check whether data-type is array or not
* typeof is very general - it returns array type as "object". Hence, to be more specific, we use Array.isArray
* array created by both [] and new Array keyword, both return true with Array.isArray(arr)
* returns a Boolean
* original array untouched  
  A computer screen shot of a code

  Description automatically generated

***What do you mean by array-like objects?***

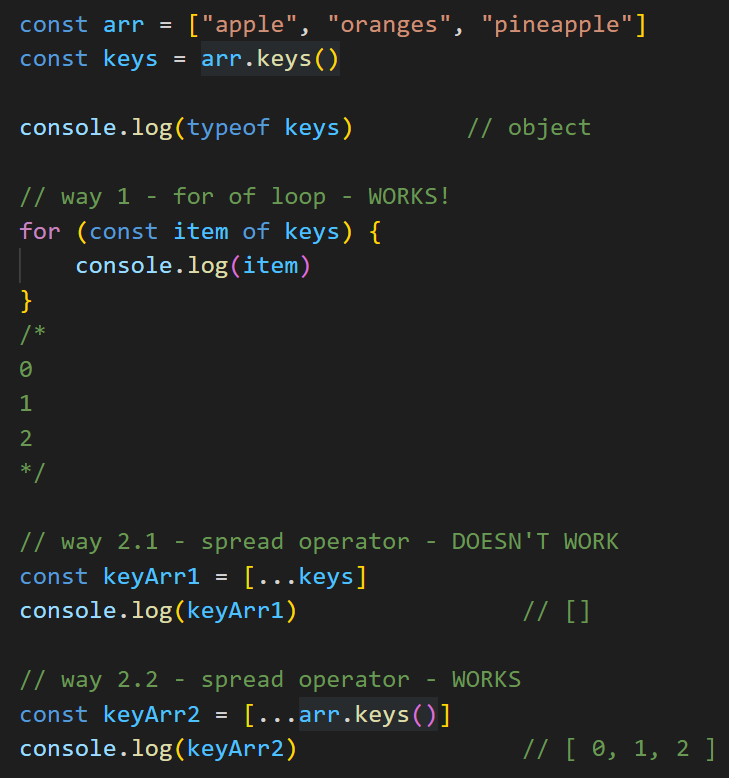
* something that has predefined length property (either predefined like string or explicitly defined)
* something that can be accessed via indexes AND has a length property (either predefined like string or explicitly defined)
* length of resultant array = length of array-like object (predefined or explicit)
* indexes need not be number of array-like object. Can be anything

**from**:

* Array.from(arraylikeObject,{callbackfn logic…})
* used to convert an array-like object to an Array
* callbackfn logic: if we want to perform some operation, before storing data into each index of resultant array
* Check the special scenario when keys of array-like objects are not numbers. Very interesting. We need to use callback function there. (otherwise, all values come as undefined)
* Also if an array-like object has only length property and no key-value pair, again we need to use callback fn (otherwise all values come as undefined)
* returns an empty array, if incoming variable is not array-like object
* returns an array, if incoming variable is array-like object
* original array untouched  
  A computer screen with colorful text

  Description automatically generated

**keys**:

* *arr.keys()*
* returns an iterable (not array) that contains all indexes of array in order
* values of iterable can be accessed via two ways:  
   a. for of loop (best way)  
   b. spread operator( there’s a catch, see snippet)
* original array untouched  
  

**values**:

* *arr.values()*
* returns an iterable (not array) that contains all values of array in order
* values of iterable can be accessed via two ways:  
   a. for of loop (best way)  
   b. spread operator( there’s a catch, see snippet)
* original array untouched  
  A screen shot of a computer program

  Description automatically generated

**entries**:

* *arr.entries()*
* returns an iterable (not array) that contains [index, value] pairs of array in order
* values of iterable can be accessed via two ways:  
   a. for of loop (best way)  
   b. spread operator( there’s a catch, see snippet)
* original array untouched  
  A screen shot of a computer program

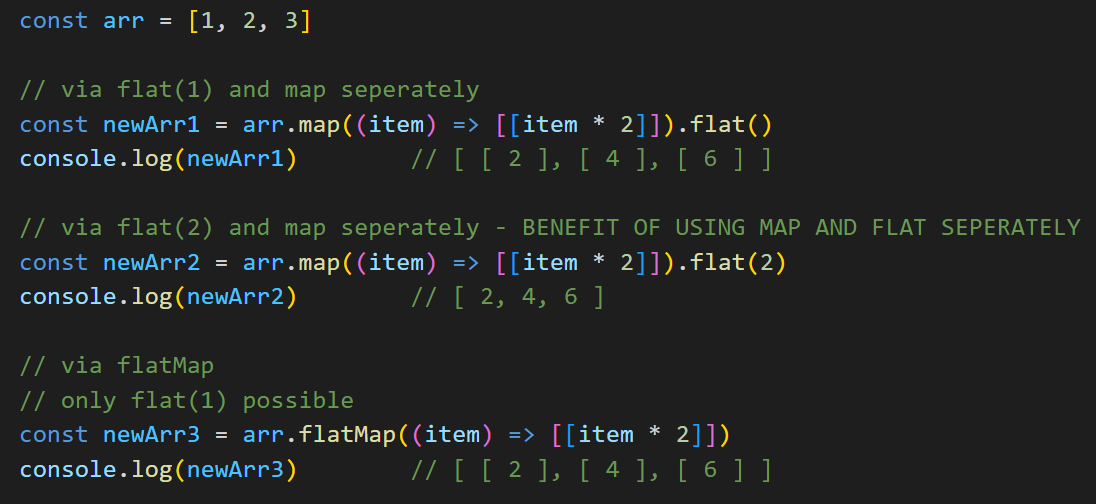
  Description automatically generated

**flat**:

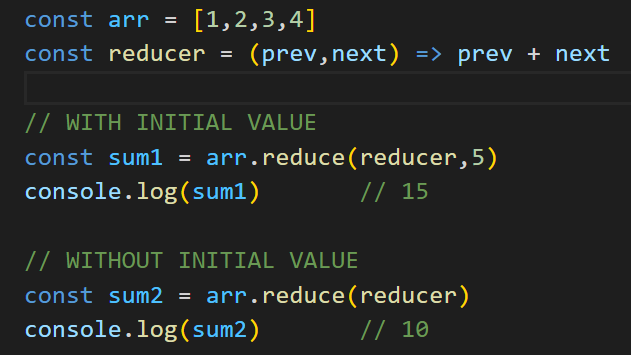
* *arr.flat(depth:optional)*
* used to flatten an array i.e. reduce dimensions of array
* original array untouched
* *depth*: how many dimensions we want to reduce of each array item and array ( if not provided, default value = 1)  
  *how flatten algo works:*
* Consider an example a flat(2) is applied on array [[[1],2]] (3d array)
* arr[0][0][0] returns value = 1 and arr[0][1] returns value = 2
* flat(2) is basically flat(1) successively twice
* FIRST flat(1): [[1],2] // 2-d array (flat execute at array level)
* SECOND flat[2): [1,2] // 1-d array (flat execute at element level as array level dim = 1)
* *Two extremists are determined:*   
   a. what the deepest element’s dimension in the array? Ans: 3 (of value 1)  
   b. what the nearest (shallow) element’ dimension in the array? Ans: 2 (of value 2)  
  (a) represents dimension of deepest array element and (b) represents dimension on array level flattening starts from (b). Starts reducing the value coming from (b) until it reaches 1. Once (b) = 1, and flat remaining, moves towards reducing (a) so flatten happens at:  
   - at array level (once = 1, moves to next step)  
   - at element level  
  A screen shot of a computer program

  Description automatically generated

**flatMap**:

* *arr.flatMap(callbackFun)*
* above is same as: arr.map(callbackFun).flat(1)
* a map method is followed by a flat(1)
* disadvantage : with flatMap, you can go down max by 1 level I.e. flat(1), since there is no way to specify a depth here
* all other points same as flat()  
  

**reduce**:

* arr.reduce(reducerFunc, initialValue:optional)
* used to perform some kind of operation on array values and yield a result [ex- sum of all elements of array etc]
* returns a primitive or non-primitive value
* original array untouched
* *how reduce works:* See example below  
    
    
    
    
  *(a) when initial value defined:*

|  |  |  |
| --- | --- | --- |
| prev | next | Result (based on logic of reducerFunc) |
| 5 (initial value) | 1 (index = 0) | 6 |
| 6 (result of previous cycle) | 2 (index = 1) | 8 |
| 8 (result of previous cycle) | 3 (index = 2) | 11 |
| 11(result of previous cycle) | 4 (index =3) | 15 (returning value) |

*(b) when initial value not defined:*

|  |  |  |
| --- | --- | --- |
| prev | next | Result (based on logic of reducerFunc) |
| 1 (index = 0) | 2 (index = 1) | 3 |
| 3 (result of previous cycle) | 3 (index = 2) | 6 |
| 6 (result of previous cycle) | 4 (index = 3) | 10 (returning value) |

* reducerFunc takes two params - prev and next
* if initial value is defined, it becomes first value of prev.
* if initial value is not defined, index = 0 becomes first value of prev
* result of first cycle becomes prev value of next cycle

**reduceRight**:

* everything same as reduce, only difference is that operation operates from last to first index  
  