# JS Collections

## Collections Basics

1. An object with identical keys is overwritten by the last appearing key.

> let obj = { fruit: 'apple', vegetable: 'carrot', fruit: 'pear' }

> obj

{ fruit: 'pear', vegetable: 'carrot' }

1. Differentiating between a non-existent property vs a property that has *undefined* as its value.
   1. ObjName.hasOwnProperty(“a”)
   2. Object.keys(ObjName).includes(“a”)

let obj = { a: 'foo', b: 'bar', c: undefined};

obj.hasOwnProperty('c'); // => true

obj.hasOwnProperty('d'); // => false

Object.keys(obj).includes('c'); // => true

Object.keys(obj).includes('d'); // => false

1. Difference between arrays and objects:
   1. Arrays use non-negative integers as its primary keys
   2. Adding elements to arrays changes the value of its length property and changing its length property causes the number of elements to change
   3. Arrays can still take in key: value pairs, however that will not change its length property. Further these key: value pairs (non-element properties) are ignored by arraymethods like: forEach, map and filter.

let arr = ['foo', 'bar', 'qux'];

arr['boo'] = 'hoo';

arr[-1] = 374;

arr; // => [ 'foo', 'bar', 'qux', boo: 'hoo', '-1': 374 ]

arr.length; // => 3 (not 5!)

arr.forEach(element => console.log(element)); // prints: foo, bar, qux

Object.keys(arr); // => [ '0', '1', '2', 'boo', '-1' ]

* 1. Using Object.keys(arr) will still return all the property names

1. Check if an obj/arr is an array: Array.isArray(objName)
2. Convert an object to array: Object.entries(objName) . Key-value pairs are returned as sub-arrays.

let obj = { sky: 'blue', grass: 'green' };

Object.entries(obj); // => [ [ 'sky', 'blue' ], [ 'grass', 'green' ] ]

1. Reassigning string characters: Since strings are immutable, you cannot use array element assignment syntax to reassign values/letters. However, if you do, you will not get any errors.  
   To make the desired changes in a string, you make a new string (as shown above).

let str = ‘bob’;

str[0] = ‘B’;

str; // => ‘bob’

str = ‘B’ + str.slice(1);

str; // => ‘Bob’

# String Methods

1. Are strings collections? : Strings aren’t really collections since you can’t store generic data into them however they have a few properties that associate them with collections:
   1. You can access individual/multiple characters of the strings.
   2. You can loop through all the characters using the length property in conjunction with for/while loops
2. Concatenating strings:
   1. Using ‘+’
   2. Using str.concat(str2, str3, …)
3. Note: Any operation performed on strings returns a new string. None of the methods that operate on strings mutate the string since in JS strings are immutable.
4. .includes(): This string method takes a str as an argument and returns a Boolean signifying whether that string exists within the string that includes was called on.  
   The optional second parameter of includes() specifies which index to start looking from in the string.

* ‘abcdef’.includes(‘b’)

true

* ‘abcdef’.includes(‘b’, 2)

false

1. .split()
2. .trim() : Removes whitespace from both ends of the string. Useful for cleaning user inputs which often contain unnecessary whitespace on either sides. The other two variations of this method are .trimStart() and .trimEnd() which remove whitespaces from start and end of string respectively.
3. .toUpperCase() and .toLowerCase()
4. .charAt() : Identical to using brackets for indexing elements from a string. Difference: When your index is out of range it returns an ‘’, on the other hand bracket notation would return undefined.

* let sentence = "It's a walk in the park."
* sentence.charAt(5)

'a'

* sentence[10]

‘k’

* sentence.charAt(25)

‘’

* sentence[25]

undefined

1. .charCodeAt() : Identical to .charAt() but instead of returning the character at given index, it returns the unicode code point (character code) of the character at that index.  
   If no index is given, it assumes a default value of index 0.

* ‘abcdef’.charCodeAt(1);

98

1. String.fromCharCode() : Is the opposite of charCodeAt(), that is given the unicode code point, it returns the character represented by that character code. Note: This is a static method (not a prototype method).
2. .endsWith(searchStr, endIndex) : Determines whether a string ends with certain characters (passed as arguments). It returns a boolean
3. .startsWith(searchStr, startIndex) :
4. .repeat(count): repeats a particular string for the provided number of counts.

# PEDAC Process

1. PEDAC
   1. Problem (understanding and describing in own words),
   2. Examples/Test cases
   3. Data Structures
   4. Algorithms
   5. Code with intent
2. Problem (Understanding)
   1. When it comes to solving complex problems, most people have sufficient knowledge of the language syntax and its functions/methods. Where they get stuck, is understanding the problem and determining an appropriate algorithm.
   2. Note: Always clarify any part of the problem that is unclear to you with the interviewer. Also verify your assumptions by looking at the given test cases / asking the interviewer.
   3. Write down the i/ps and o/ps of the problem.
   4. Identify and write all the explicit requirements as well as the implicit requirements (derived from analysis of test or edge cases) in the problem.
3. Data Structure / Algorithms
   1. Since the selection of data structure influences the algorithm, these two steps are often combined.
   2. Creating a mental model of how the algorithm works is the most important thing. Implementing it thereafter becomes pretty straightforward.
   3. Your algorithm should contain high level implementation details, especially for the “hard” parts. Trivial steps like storing in a variable, loop, condition etc. can be skipped.
   4. Initially your algorithm should only contain abstract and high-level details. These steps can then later be further broken down and filled with details as needed.
4. Code implementation:
   1. First, you must be able to write a plain English solution to the problem.
   2. Followed by formal pseudocode (optional)
   3. Finally, the language specific code.
5. Testing Frequently: Although not a part of the PEDAC approach, you must practice testing your code (very intermittently) as you write it.
6. PEDAC is not a linear process, so moving back to say the understanding stage again from algorithm stage for improvement, is normal.

# Example PEDAC process

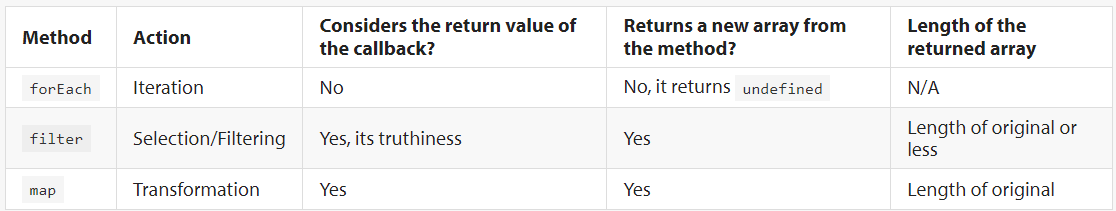
1. input: array of strings
2. output: array of strings (in descending order by max adjacent consonants)
3. rules:
   1. explicit:
      1. strings in the array must be sorted by max number of adjacent consonants (descending order)
      2. if two strings have same no. of adjacent consonants, their original order must be maintained
      3. If two consonants are separated by a space, they are considered adjacent
      4. string can contain multiple words
   2. implicit:
      1. single consonants are considered as 0 adjacent consonants, ex: baa and aa both have 0 adjacent consonants.
      2. we mutate and return the same array
4. DS:
   1. i/p and o/p are both arrays

# Selection and Transformation

1. selection: picking elements out of a collection based on one or more criteria. Ex: selecting all the prime numbers out of an array of numbers.
2. transformation: manipulating every element in the collection. Ex: Increment all numbers in an array by 1.
3. Using just 3 actions: Iteration, selection and transformation, we can manipulate a collection any way we need to. These actions can be packaged into a function in order to perform them on any collection.
4. Pay attention to when original collection is mutated vs when a new collection is returned. This can be a source of a lot of misunderstandings.
5. We can further make these functions more generic by adding additional parameters to them that specify the criterion for selection/transformation.
6. Its common to chain actions on collections, just make sure if an empty collection or undefined is not being returned anywhere in the chain.

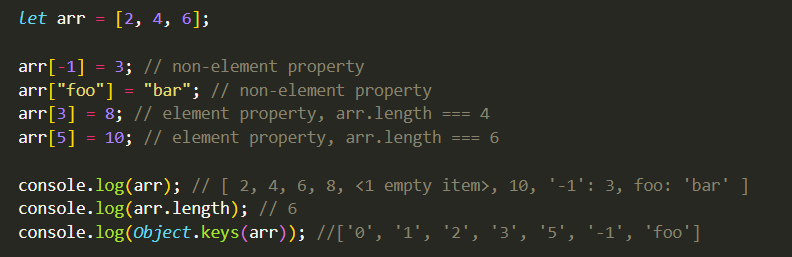
# Array Methods

1. JS provides a better option to for and while loops for iterating over collections: The Array methods **forEach()**, **filter()** and **map()**.
2. Unfortunately, these methods do not work on collection / collection-like types like Objects or strings. However, we can convert Objects to Arrays using the methods: **Object.keys()**, **Object.values()** and **Object.entries()**.

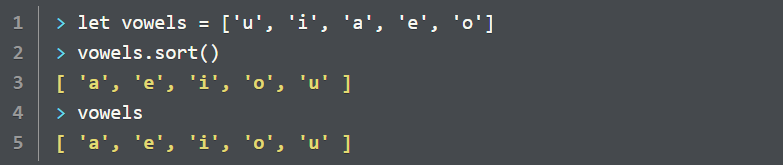
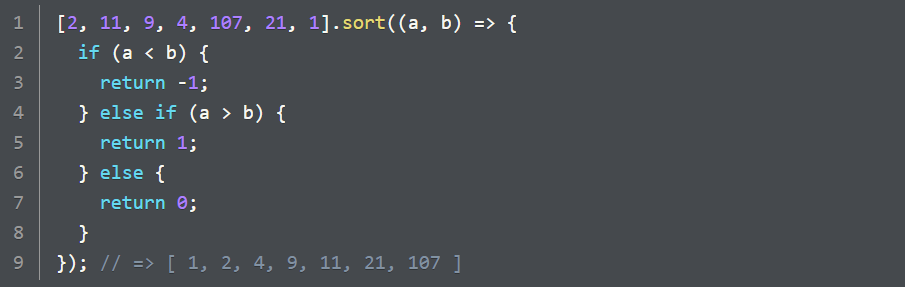
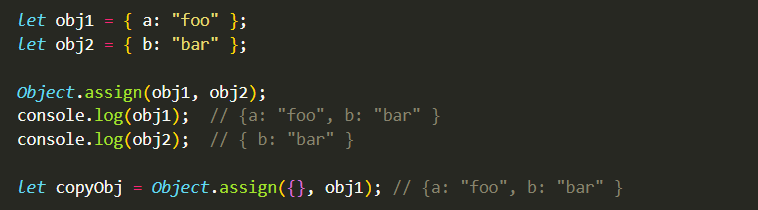
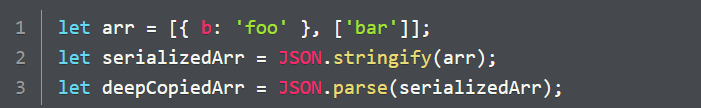


1. **some() :** If the callback fn returns a truthy value for even 1 element of the collection, the method returns true, else it returns false.
2. **every() :** If the callback fn returns a truthy value for each and every element of the collection, the method returns true, else it returns false.
3. **find() :** Also takes a callback fn as an argument and returns the first element (from the array) that returns a truthy value. If no truthy value is returned for any of the elements, the method returns undefined.
4. **findIndex() :** Same as find() but returns the index for the element that returnsa truthy value. Also it returns -1 instead of undefined, if no truthy value is returned for any of the elements.
5. **reverse() :** Reverses the array (in-place). Hence the original array gets mutated.
6. **includes() :** Searches array for an exact match of the element (like ===). However, it can also look for NaN value even though NaN ==/=== NaN returns false.

# Arrays: What is an element?

1. Indices of array are also property names of the array object (translated to string).
2. However, not all properties of the array are elements of the array (contribute to the length of array).
3. In order for a property to be contributed to length of array, the property must be a non-negative integer. A property that is any other value is not considered an array element. Ex: -3 and foo are not array elements (even though they are array properties).  
   
4. All the array methods also ignore non-element properties. Ex: Calling map() on arr will not include -1 and foo.

# Sorting

1. sort() is a destructive fn. It does not return a new sorted array. In fact, it mutates the original array and returns a reference to the original array.  
   
2. For this reason, it is better to make a copy of the array before sorting it. We can do this by using slice() as in: vowels.slice().sort()
3. **sort()** **by default converts all data types (numbers, objects etc.) to a** **string** before comparing them. For any other comparison, it takes callback function as argument. The return value of this fn determines the final sequence produced by sort().  
   
4. Rules for sorting:
   1. If callback fn returns a number less than zero, then a comes before b.
   2. If callback fn returns a number greater than zero, then b comes before a.
   3. If callback fn returns a number equal to 0, then order remains unchanged.
5. Copying (shallow) objects:
   1. Arrays
      1. Using slice(): arr.slice()
      2. Using spread operator: […arr]
   2. Objects
      1. Using: Object.assign({}, obj)
   3. Note: These methods create a shallow copy, that is changes made within the nested objects of the copy will be reflected in original as well.
6. Deep Copy:
   1. JS doesn’t have an explicit method/fn for deep copying objects, however there is an indirect way to do it (By stringifying/serializing the object and then parsing it, all using JSON).   
      
   2. This method only works for nested arrays and objects only, you cannot use it for objects that contain methods, complex objects like dates or custom defined objects.
7. Freezing Objects (Using **Object.freeze()**):
   1. This method prevents objects from being modified. However, the nested objects can still be mutated.
   2. Once frozen, the object cannot be mutated. It throws a TypeError if attempted to be mutated or assigned.
   3. Only objects can be frozen since primitives are already unmutable/frozen.
   4. We can check whether an object is frozen, using **Object.isFrozen()**
   5. Remember that these methods only freeze objects passed to them, not the objects nested within them (shallow freezing).

# Functions as first-class objects and Higher Order Functions

1. In most languages, the term first-class object/value is used to describe values that meet the following properties:
   1. Can be assigned to a variable or an element of a   
      data structure (like array objects etc.)

Functions that take in other fns as arguments and/or return a function as value are called Higher Order Functions (HOF)

* 1. Can be passed as an argument to a function
  2. Can be returned as the return value of a function

1. In JS primitive values, arrays, objects and functions all meet these criteria.
2. First-class functions (Ex: methods using callback functions like forEach, map, filter etc.) allows us to write code more declaratively rather than imperatively (like using conventional loops).
3. In declarative approach we describe to the machine what needs to be done rather than a step-by-step account on how it should be done.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Action | Performed on | side effect | return value | is return value used? |
| method call (map) | outer array | none | New array ([1, 3]) | No |
| callback execution | each sub-array | none | Numbers: 1 and 3 | Yes, used by map for transformation |
| element reference (arr[0]) | each sub-array | none | element at 0th index of each sub-array | Yes, used by method console.log |
| method call (console.log) | element at 0th index of each sub-array | outputs a string representation of the integer | undefined | No |
| element reference (arr[0]) | each sub-array | none | numbers 1 and 3 | yes, explicitly returned by the callback |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Action | Performed on | side effect | return value | is return value used? |
| outer method call (map) | outer array | none | New outer transformed array | No |
| outer callback execution | each sub-array | none | Subarrays [2, 4], [6, 8] | Yes, used by outer map for transformation |
| inner method call (map) | each sub-array | none | Subarrays [2, 4], [6, 8] | Yes, used by outer callback execution |
| inner callback execution | each number of the sub-array | none | multiplied numbers | Yes, used by inner map for transformation |
| multiplication with 2 | each number of sub-array | none | multiplied numbers | yes, explicitly returned by the callback |

# Node Debugger

|  |  |  |
| --- | --- | --- |
| Sr. | Command | Description |
| 1. | node inspect file.js | Start a node debugger (from the terminal). |
| 2. | exec varName | To get a variable value, evaluate an expression or execute a function. |
| 3. | n / next | go to the next line of code |
| 4. | c / cont | run the code until a break (debugger statement) is encountered (or until the end) |
| 5. | run / restart | inspect the code again |
| 6. | debugger | statement that depicts a “break” while inspecting the code |
| 7. | repl | opens up an repl |
| 8. | .exit / ctrl + c | exit the repl / node inspect |
|  | Adding and Removing breakpoints within the debugger itself while it is still running | |
| 9. | setBreakpoint() / sb() | set breakpoint on the current line OR pass the line number as an argument to set on the specified line |
| 10. | clearBreakpoint() / cb() | takes two arguments: name of the file and line number |
| 11. | list() | lists down 5 lines of code (by default) does not execute them though. Takes in an argument for number of lines to display. |
| 12. | s / step | In case of callback fn / inner fn, this command allows you to step into the execution of this inner fn (to see what’s happening inside) rather than moving over to the next expression or function call. |
| 13. | o / out | step outside of the inner function. |
| 14. | help | lists all the commands that are available (for node debugger) |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| function | input | output |
| initializeDeck() | - | deck (nested array) |
| shuffle(deck) | deck | deck |
| total() | subset of deck | number |
| busted() | subset of deck | Boolean |
| playerTurn() | subset of deck | - |
| dealerTurn() | subset of deck | - |
| compareCards() | subset of deck, subset of deck | string |
| displayResult() | subset of deck, subset of deck | - |
| displayPlayerCards(),  displayDealerCards() |  |  |
|  |  |  |
|  |  |  |

1. initialize and shuffle the deck
2. distribute 2 cards each to player and dealer, display 3 cards (hide 1 dealer card)
3. player turn:
   1. Ask player: stay or hit
4. dealer turn
5. display final result

|  |  |  |
| --- | --- | --- |
| Sr. | Step | Functions |
| 1. | initialize and shuffle the deck | initializeDeck(), shuffleDeck() |
| 2. | distribute 2 cards each to player and dealer, display 3 cards (hide 1 dealer card) | pickCard(), displayCards() |
| 3. | player turn: | playerTurn(), displayPlayerCards() |
| 4. | Ask player: stay or hit | playerTurn() |
| 5. | dealer turn | dealerTurn(), total(), busted(), displayDealerCards() |
| 6. | final result | displayResult() |
|  |  |  |
|  |  |  |