JavaScript – W2S1
Arrays

Cyrille Jegourel – Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

Outline (Week 2, Session 1)

- The Array object
- Properties and Methods of arrays
- Shallow and deep copies of arrays

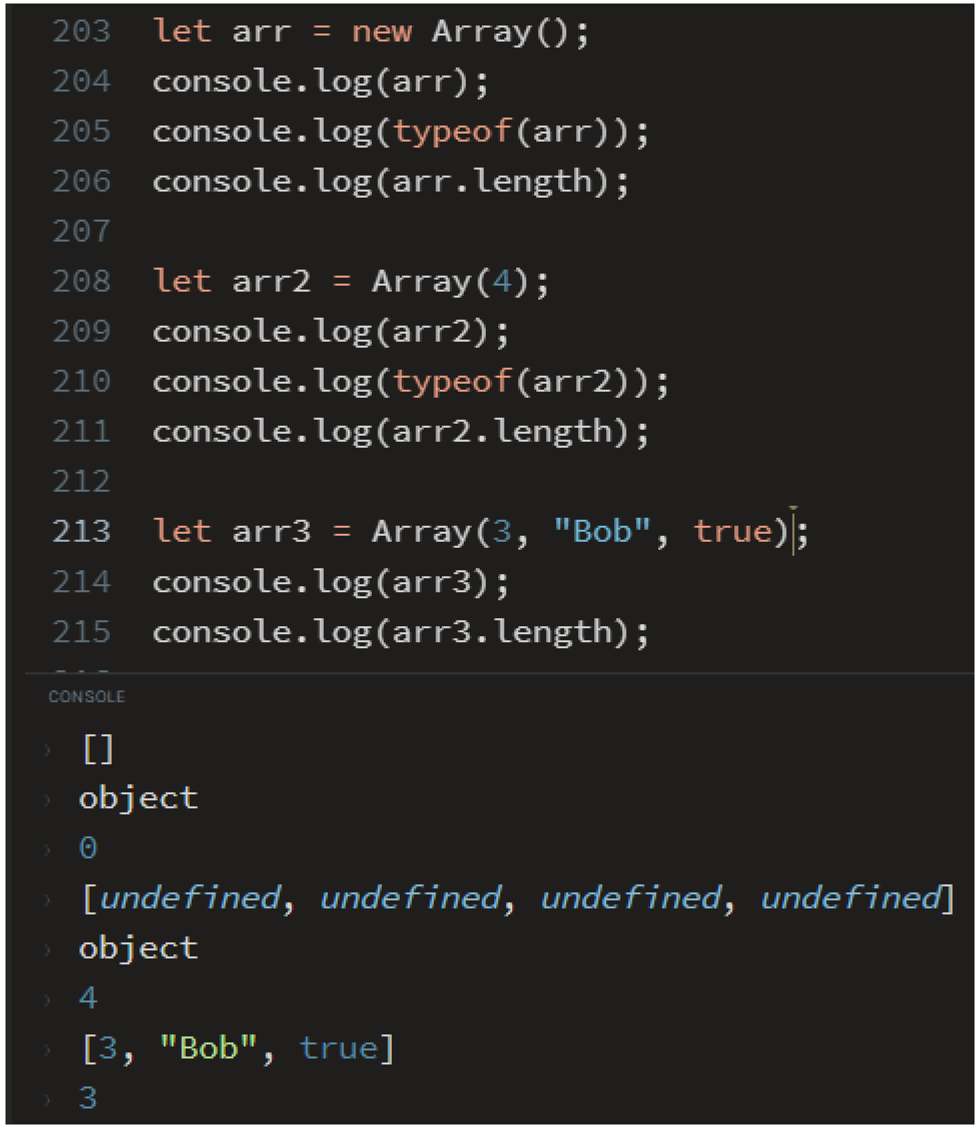
Introduction to arrays

About arrays

- An array is a way of storing a list of data.
- These data can be then accessed through their index number, beginning at 0.
- Arrays are one of the most versatile data structure available in JS.
- Arrays can contain mixed types of objects.
- You can name an array using the same rules than earlier.

Creating arrays: the array constructor

- Two ways to create arrays in JS: the array constructor **new** and array literal notation **[]**.
- To define an array with the constructor: just write **new** followed by **Array();**
- The type of an array in JS is ***object***. More on that later.
- To specify the length (e.g. 4), you can pass the number argument to Array(4).
- By default, all the data are of type undefined.
- If Array() is provided with more than one argument, all of them will be used respectively as item values of the array.
- To check the length of an array (e.g. named arr), you can use arr.length (more on methods and properties later).



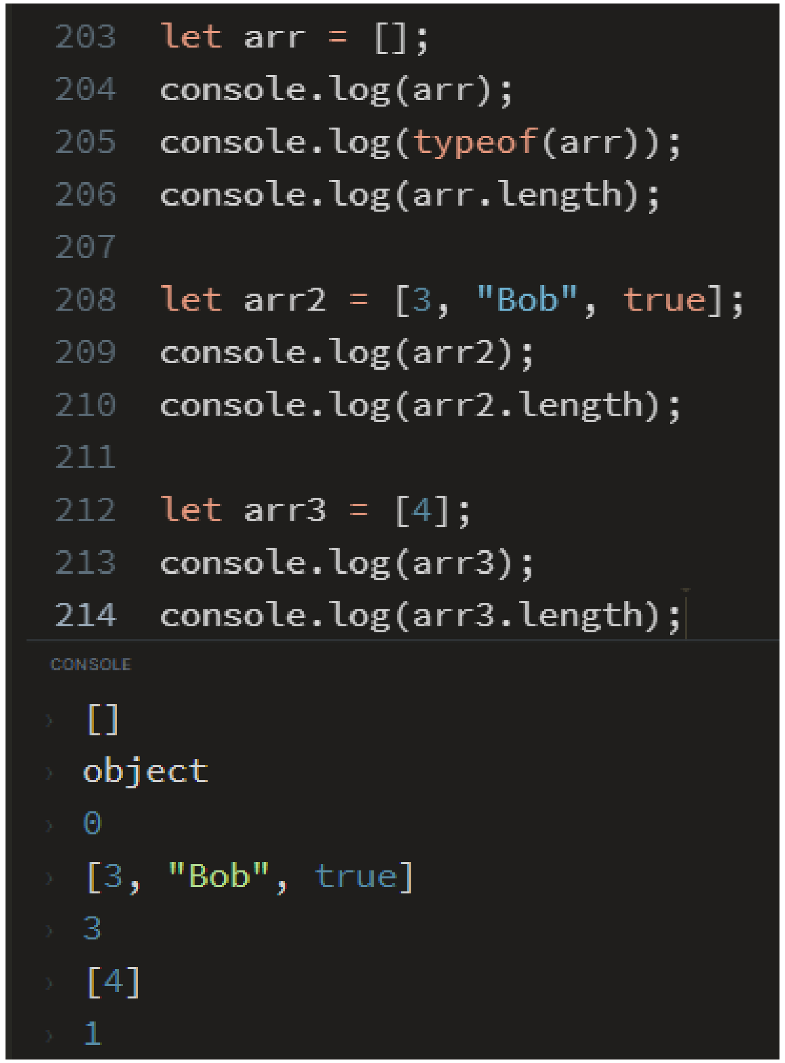
```
let arr = new Array();
console.log(arr);
console.log(typeof(arr));
console.log(arr.length);

let arr2 = Array(4);
console.log(arr2);
console.log(typeof(arr2));
console.log(arr2.length);

let arr3 = Array(3, "Bob", true);
console.log(arr3);
console.log(arr3.length);
```

CONSOLE

```
[]
object
0
[undefined, undefined, undefined, undefined]
object
4
[3, "Bob", true]
3
```

Creating arrays: the array literal notation

- Array literal notation [] provides a shorter way to define an array.
- You can just add items in the array by providing a comma-separated list of values enclosed by square brackets.
- Careful: unlike let arr = new Array(4) which defines a list of length 0 and will be able to store 4 data values, let arr = [4] defines a list of length 1 with item value 4.



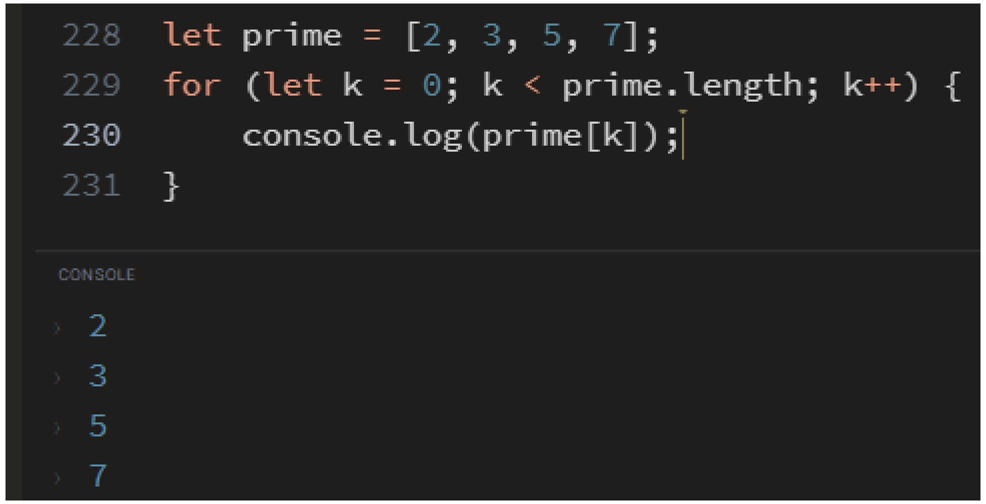
```javascript
let arr = [];
console.log(arr);
console.log(typeof(arr));
console.log(arr.length);

let arr2 = [3, "Bob", true];
console.log(arr2);
console.log(arr2.length);

let arr3 = [4];
console.log(arr3);
console.log(arr3.length);
```

CONSOLE

```
[]
object
0
[3, "Bob", true]
3
[4]
1
```

Accessing Array’s elements

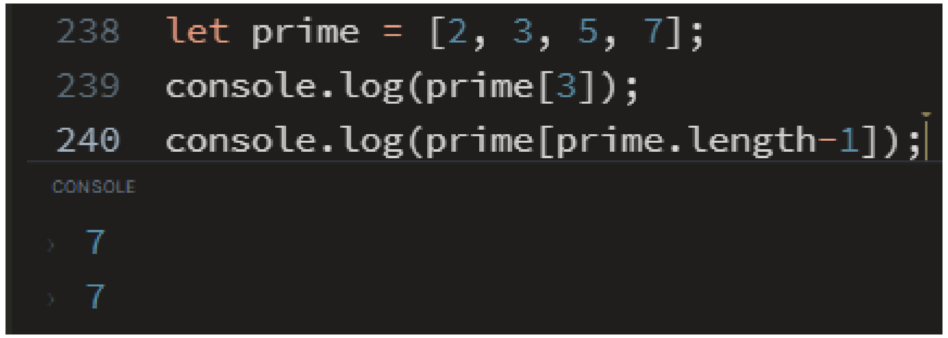
- You can access the elements of an array by using their index:
- E.g., given arr = [2, 3, 5, 7], arr[0] refers to the first element, arr[3] to the fourth or last element.
- You can use loops and the length property over arrays, e.g., to traverse them.
- Note that you can access the last element by using the length property minus 1.



```
let prime = [2, 3, 5, 7];
for (let k = 0; k < prime.length; k++) {
    console.log(prime[k]);
}
```

CONSOLE

```
2
3
5
7
```



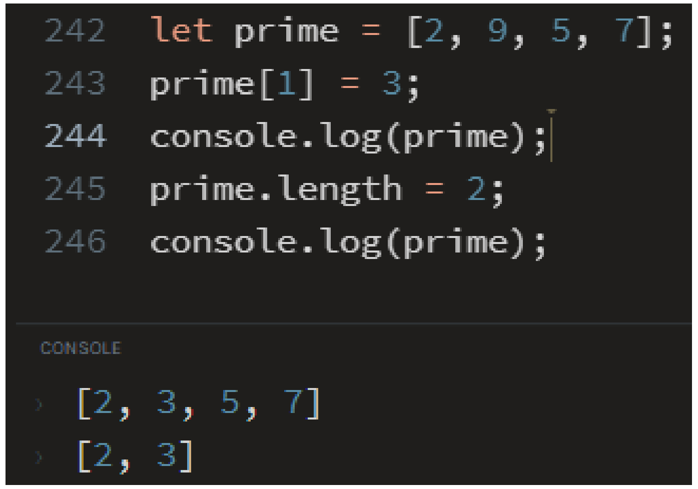
```
let prime = [2, 3, 5, 7];
console.log(prime[3]);
console.log(prime[prime.length-1]);
```

CONSOLE

```
7
7
```

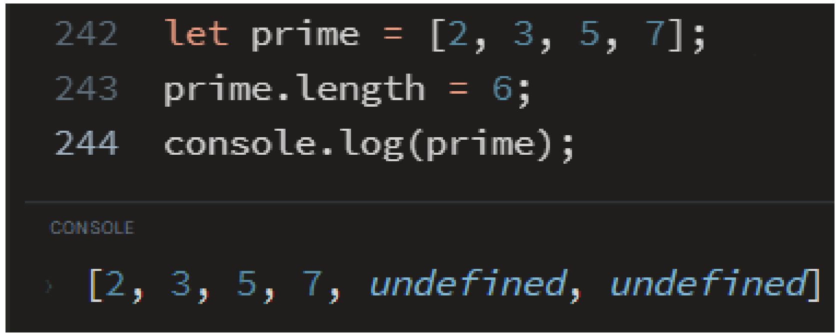
Updating an Array

- You can update the values of an array or even change its length.
- Updating an array value:
    - E.g., let prime = [2, 3, 4, 7];
    - prime[2] = 5 will replace the old value of the 3rd element, 4, by 5.
- Changing the length:
    - Initially, prime has 4 elements.
    - After executing prime.length = 2, prime has only 2 elements (2 and 3).
    - After executing prime.length = 6, prime has now 6 elements. The new elements are added to the end and are undefined by default.
    - Another way is to assign a value to an index greater than the length of the array. The elements in between will be set to undefined by default.



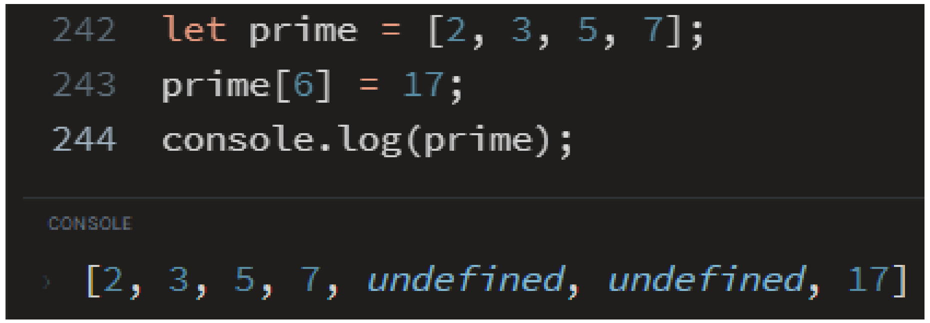
```
let prime = [2, 9, 5, 7];
prime[1] = 3;
console.log(prime);
prime.length = 2;
console.log(prime);
```

```
CONSOLE
[2, 3, 5, 7]
[2, 3]
```



```
let prime = [2, 3, 5, 7];
prime.length = 6;
console.log(prime);
```

```
CONSOLE
[2, 3, 5, 7, undefined, undefined]
```

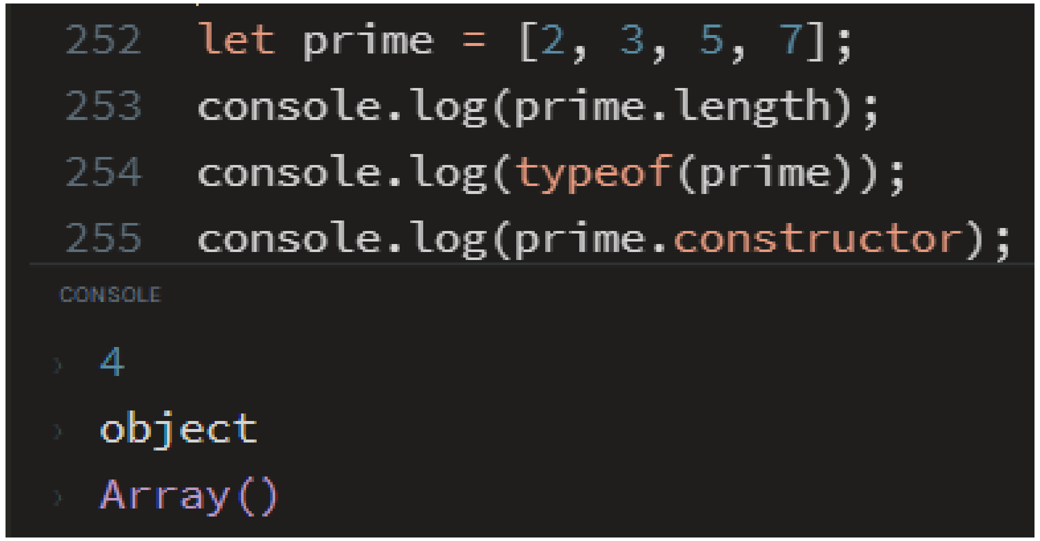


```
let prime = [2, 3, 5, 7];
prime[6] = 17;
console.log(prime);
```

```
CONSOLE
[2, 3, 5, 7, undefined, undefined, 17]
```

Array properties

- Arrays come with several properties. You already know **length**.
- Here are a few others:
    - **constructor, index, input and prototype**.
    - More on the three last when you will be familiar with *regular expressions.*
    - **arr.constructor** refers to the constructor function used to create an instance of arr.
    - Note: in JS, there are many different types of objects (arrays, documents, etc.): **typeof(arr)** returns **object** which is vague but **arr.constructor** is more precise and returns **Array()**.



```
let prime = [2, 3, 5, 7];
console.log(prime.length);
console.log(typeof(prime));
console.log(prime.constructor);
```

CONSOLE

```
4
object
Array()
```

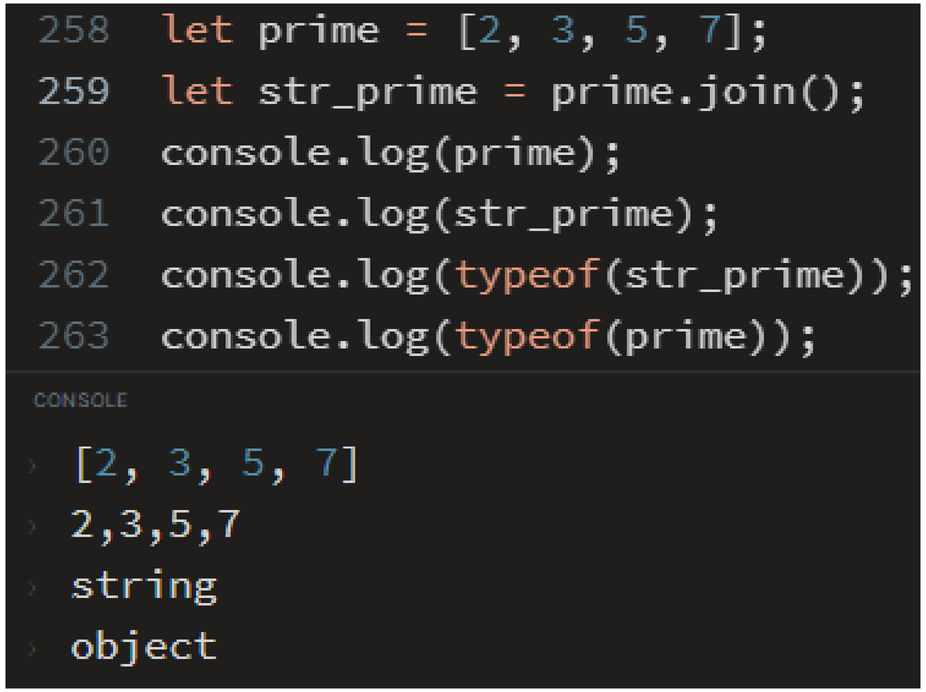
Array Methods

- Methods are functions which can be used using the dot notation on specific objects.
- In the next slides, we will cover the most important methods of Array objects.

The join() method

- Used to combine the items of an array into a string, each item being separated by a comma by default, or a string given as an argument.

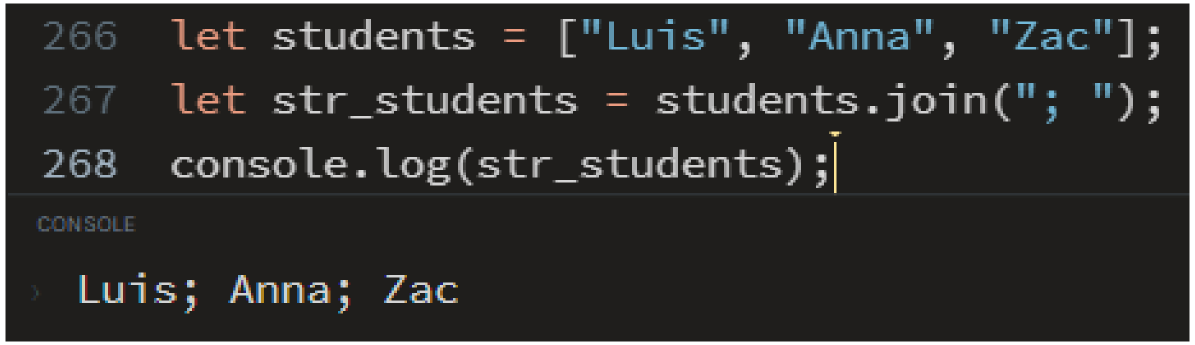
- In the 2nd example, the three names will be separated by a semi-colon followed by one space.



```
let prime = [2, 3, 5, 7];
let str_prime = prime.join();
console.log(prime);
console.log(str_prime);
console.log(typeof(str_prime));
console.log(typeof(prime));
```

CONSOLE

```
[2, 3, 5, 7]
2,3,5,7
string
object
```



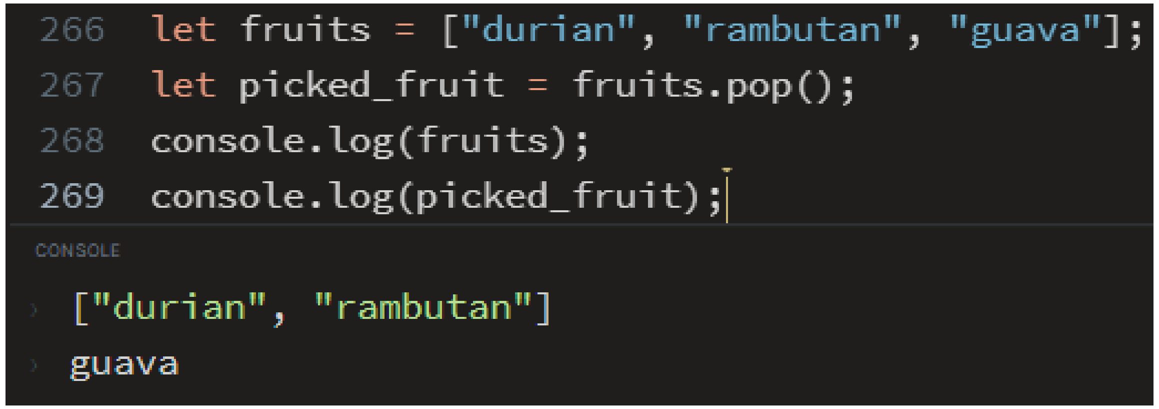
```
let students = ["Luis", "Anna", "Zac"];
let str_students = students.join("; ");
console.log(str_students);
```

CONSOLE

```
Luis; Anna; Zac
```

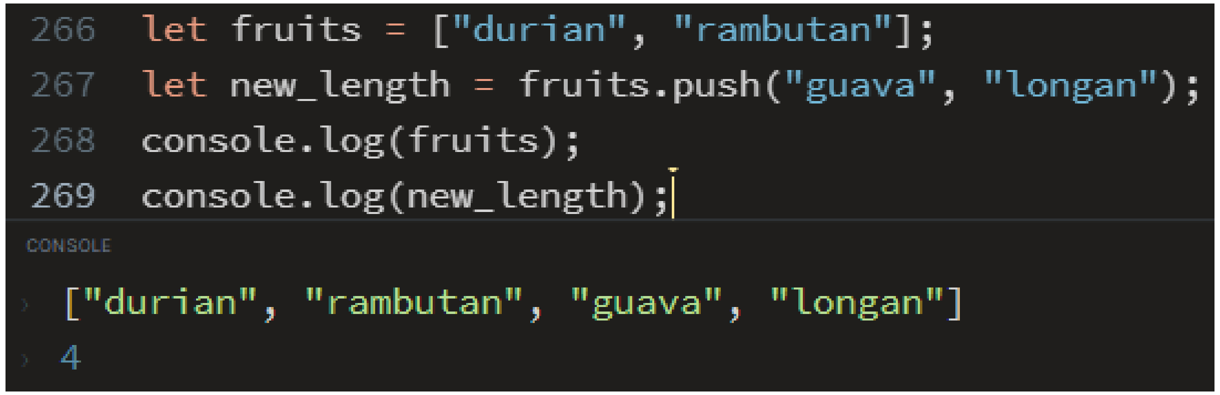
The pop and push() methods

- **pop()** is used to remove the last item of an array.
- The removed item can be returned and assigned to a variable for later use.
- **push()** is used to add one or more elements to the end of an array.
- The new length of the array can be returned by **push()** by assigning the result to a variable.



```
let fruits = ["durian", "rambutan", "guava"];
let picked_fruit = fruits.pop();
console.log(fruits);
console.log(picked_fruit);
```

```
CONSOLE
› ["durian", "rambutan"]
› guava
```

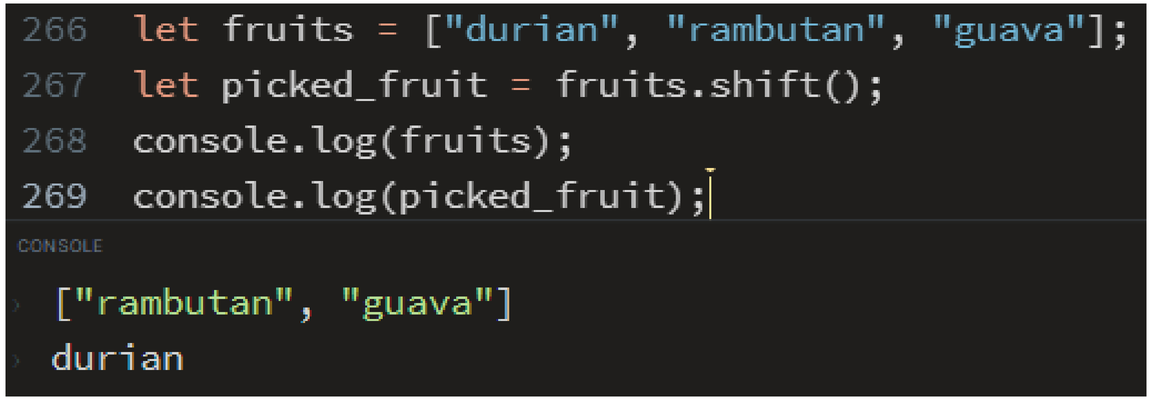


```
let fruits = ["durian", "rambutan"];
let new_length = fruits.push("guava", "longan");
console.log(fruits);
console.log(new_length);
```

```
CONSOLE
› ["durian", "rambutan", "guava", "longan"]
› 4
```

The shift() and unshift() methods

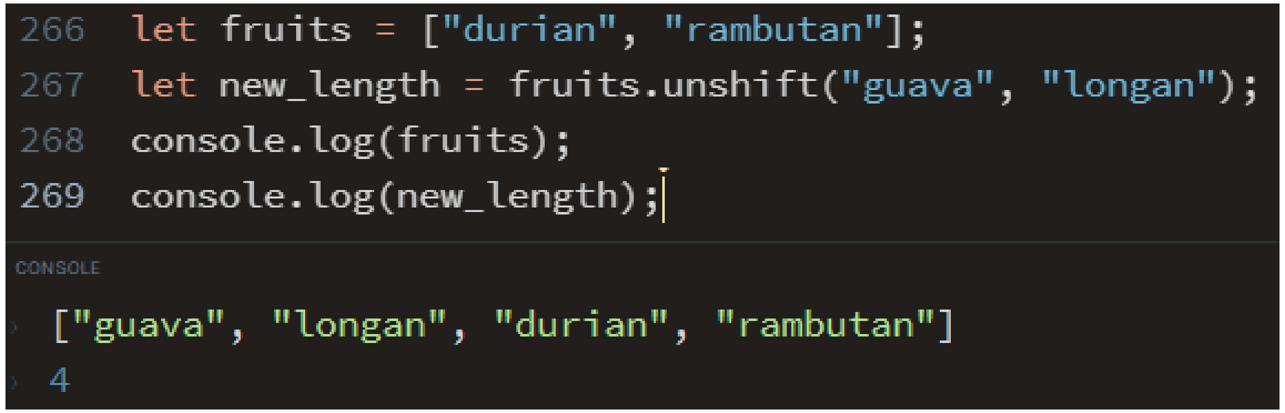
- **shift()** and **unshift()** are respectively similar to **pop()** and **push()**, except that they remove elements from or add elements to the beginning of the array.
- Note: **push()** and **pop()** treat an array like a stack (last in, first out) whereas **shift()** and **unshift()** treat an array like a queue (first in, first out).



```
let fruits = ["durian", "rambutan", "guava"];
let picked_fruit = fruits.shift();
console.log(fruits);
console.log(picked_fruit);
```

CONSOLE

```
["rambutan", "guava"]
durian
```



```
let fruits = ["durian", "rambutan"];
let new_length = fruits.unshift("guava", "longan");
console.log(fruits);
console.log(new_length);
```

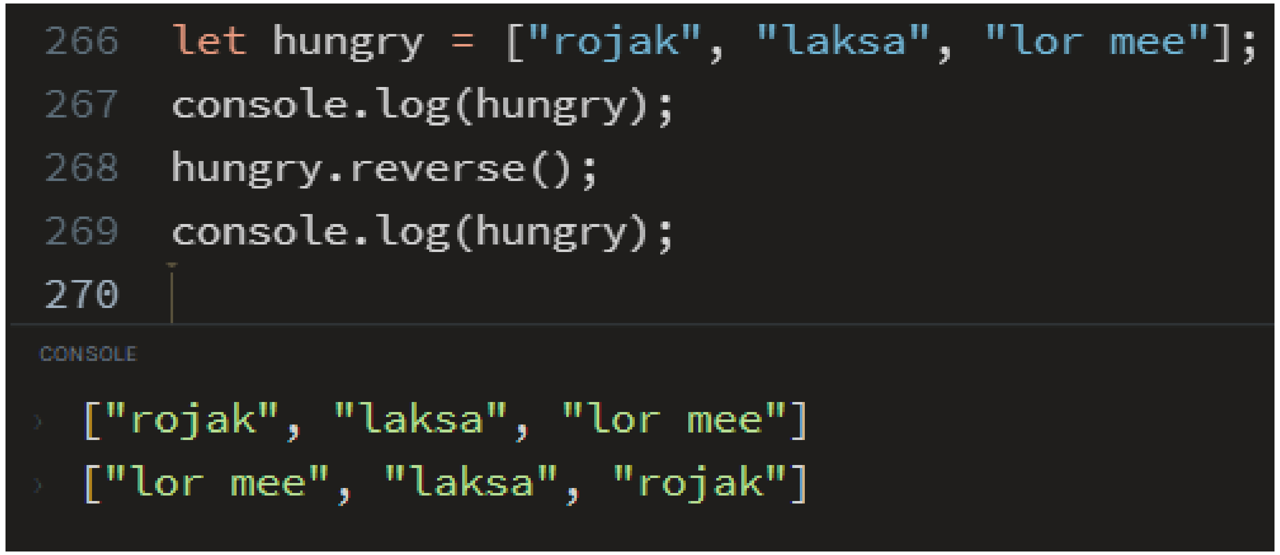
CONSOLE

```
["guava", "longan", "durian", "rambutan"]
4
```

The reverse() method

- Used to reverse the order of the elements of the array.

- Note that it alters the original array, rather than returning a new array.



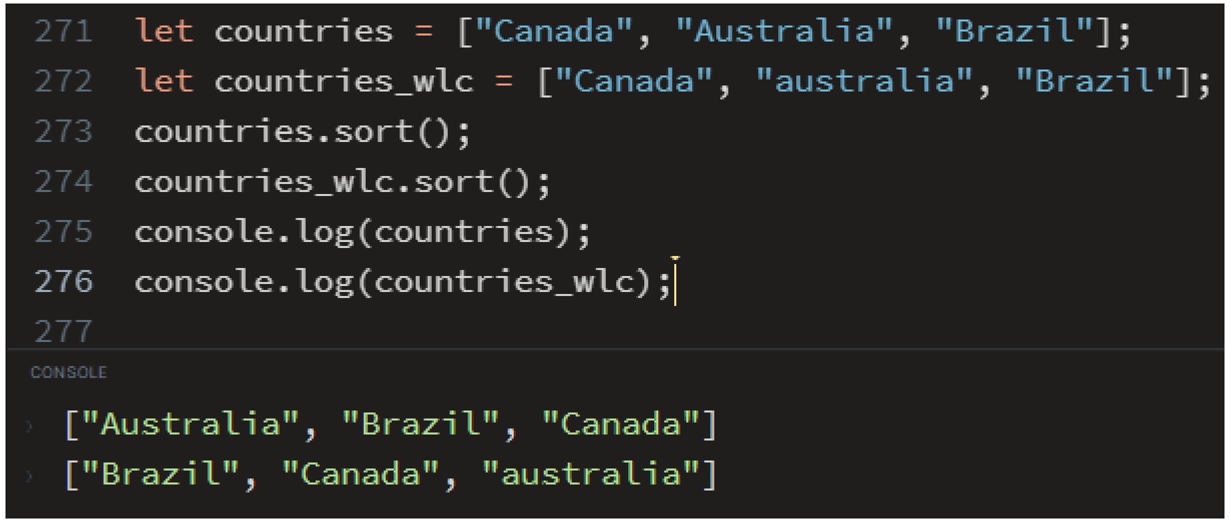
```
let hungry = ["rojak", "laksa", "lor mee"];
console.log(hungry);
hungry.reverse();
console.log(hungry);

```

CONSOLE

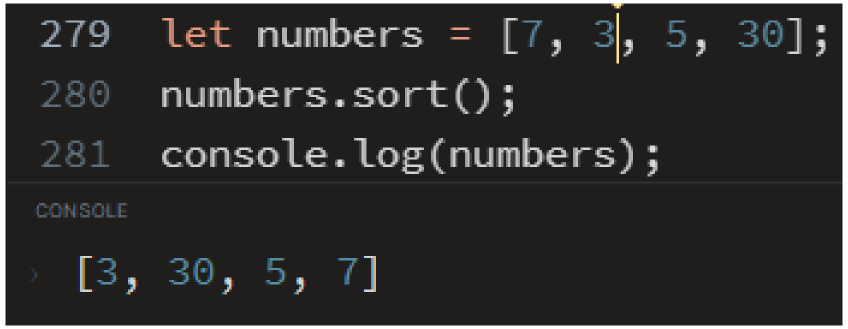
```
["rojak", "laksa", "lor mee"]
["lor mee", "laksa", "rojak"]
```

The sort() method

- The **sort()** method converts each element into a string and then sorts them by ASCII alphabetic order.
- **Caution**: be careful with lowercase and uppercase letters, but also with arrays of mixed-type elements or with numeric arrays…
- Numbers have lower rank than uppercase letters, which have lower rank than lowercase letters.
- **Caution**: Numbers are also sorted by “alphabetic order”!



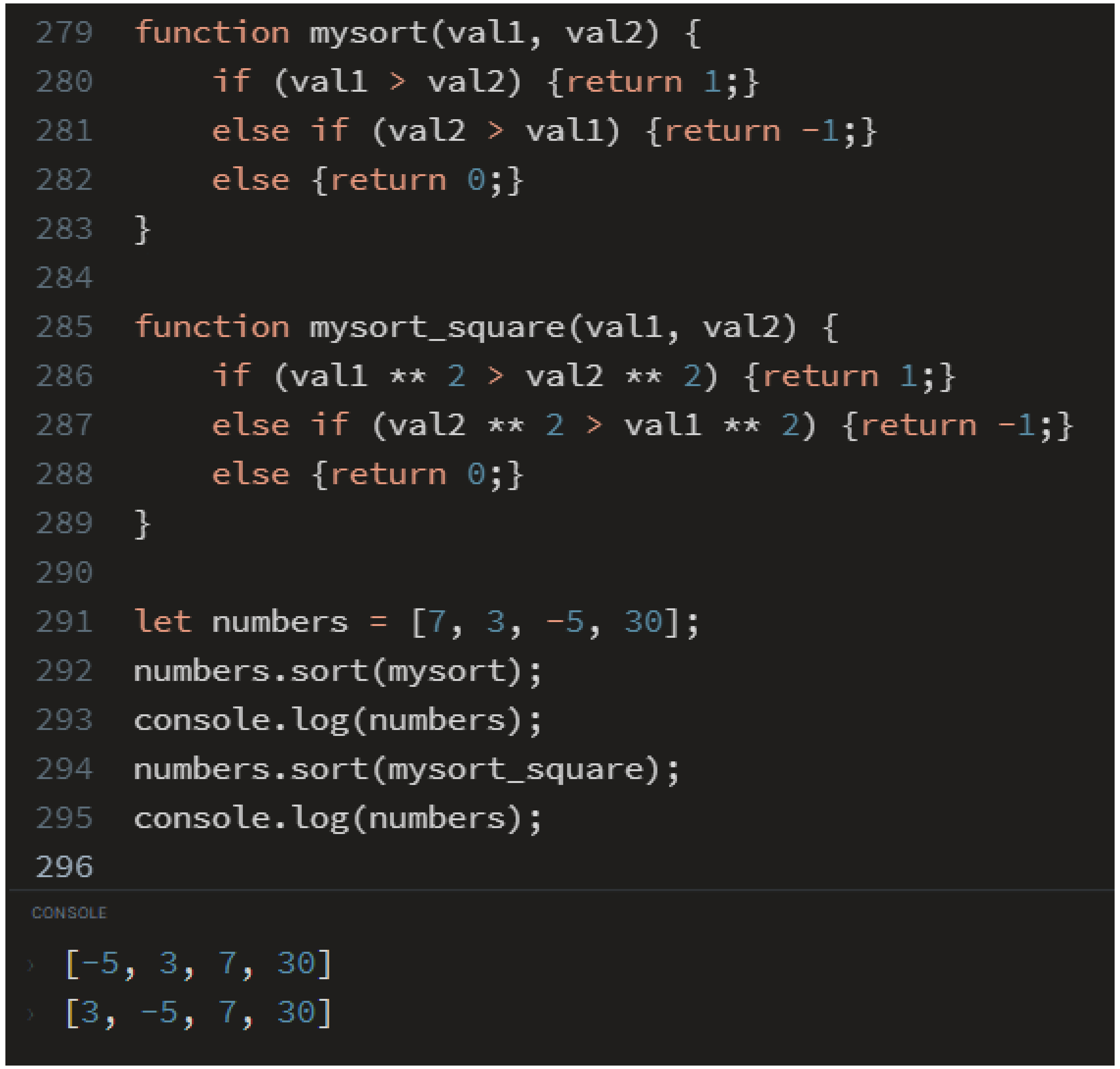
```
let countries = ["Canada", "Australia", "Brazil"];
let countries_wlc = ["Canada", "australia", "Brazil"];
countries.sort();
countries_wlc.sort();
console.log(countries);
console.log(countries_wlc);

CONSOLE
› ["Australia", "Brazil", "Canada"]
› ["Brazil", "Canada", "australia"]
```



```
let numbers = [7, 3, 5, 30];
numbers.sort();
console.log(numbers);
CONSOLE
› [3, 30, 5, 7]
```

Sorting numeric arrays

- To sort a numeric array, you first need to define a comparison function.
- A comparison function is a function that, given two arguments, returns a positive number, a negative number or zero based on the result of comparing the two argument values.
- A numeric array can be then sorted using this function name as an argument of the sort() method.
- **Remain cautious when an array contains mixed-type elements!**

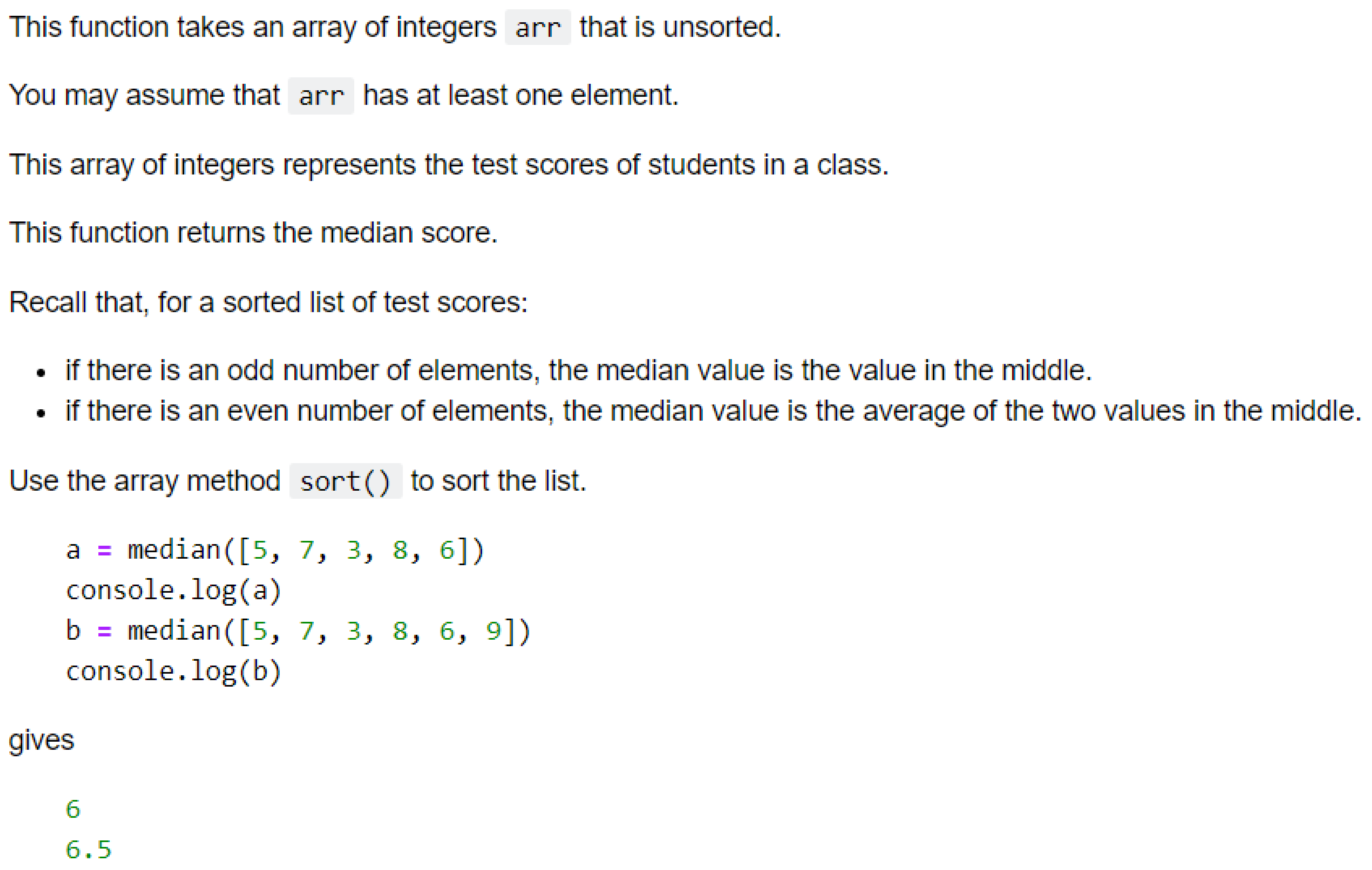


```
function mysort(val1, val2) {
    if (val1 > val2) {return 1;}
    else if (val2 > val1) {return -1;}
    else {return 0;}
}

function mysort_square(val1, val2) {
    if (val1 ** 2 > val2 ** 2) {return 1;}
    else if (val2 ** 2 > val1 ** 2) {return -1;}
    else {return 0;}
}

let numbers = [7, 3, -5, 30];
numbers.sort(mysort);
console.log(numbers);
numbers.sort(mysort_square);
console.log(numbers);

```

CONSOLE

```
[-5, 3, 7, 30]
[3, -5, 7, 30]
```

Activity 1: median(arr)



This function takes an array of integers `arr` that is unsorted.

You may assume that `arr` has at least one element.

This array of integers represents the test scores of students in a class.

This function returns the median score.

Recall that, for a sorted list of test scores:

- if there is an odd number of elements, the median value is the value in the middle.
- if there is an even number of elements, the median value is the average of the two values in the middle.

Use the array method `sort()` to sort the list.

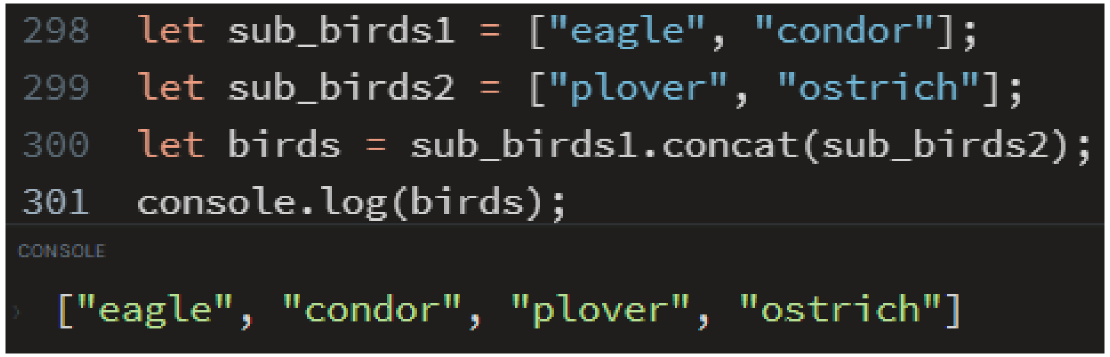
```
a = median([5, 7, 3, 8, 6])
console.log(a)
b = median([5, 7, 3, 8, 6, 9])
console.log(b)
```

gives

```
6
6.5
```

The concat() method

- Concatenates the array with its arguments and returns a new array containing all the elements in order.
- It does not alter the original array.
- You can concatenate more than two arrays and objects of different types.

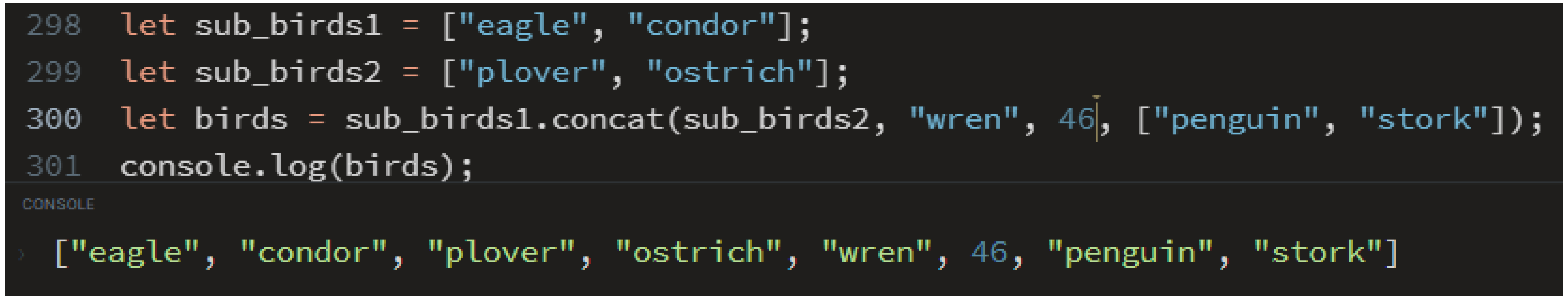


```
let sub_birds1 = ["eagle", "condor"];
let sub_birds2 = ["plover", "ostrich"];
let birds = sub_birds1.concat(sub_birds2);
console.log(birds);
```

CONSOLE

```
["eagle", "condor", "plover", "ostrich"]
```

- Note that every array argument is flattened in the concatenation.



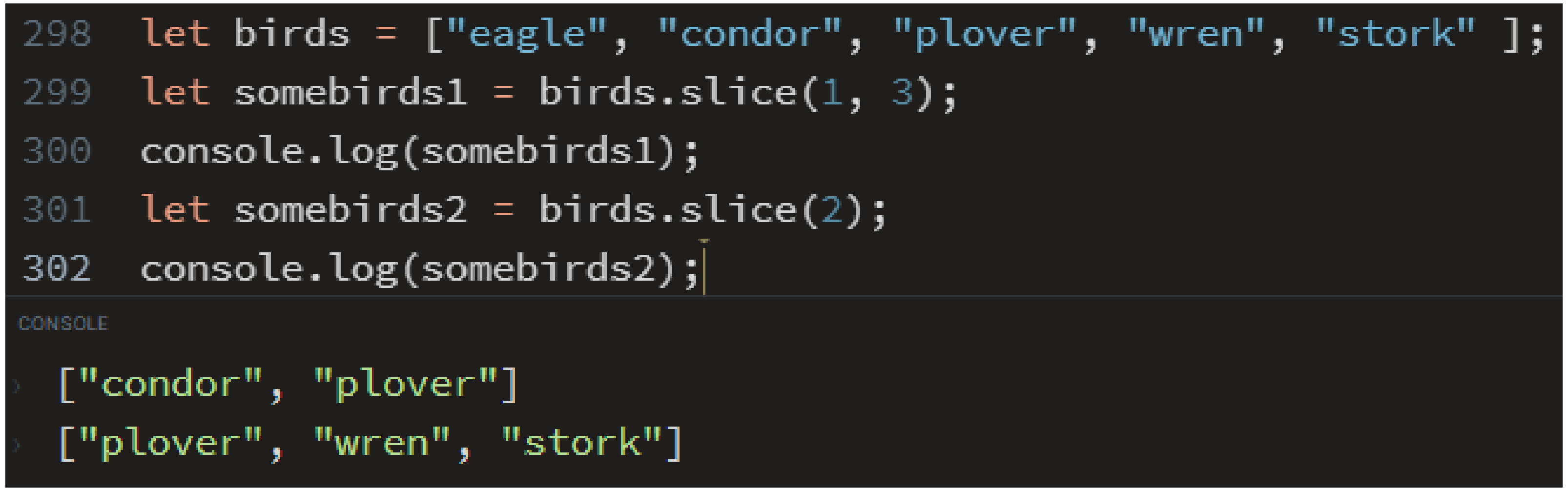
```
let sub_birds1 = ["eagle", "condor"];
let sub_birds2 = ["plover", "ostrich"];
let birds = sub_birds1.concat(sub_birds2, "wren", 46, ["penguin", "stork"]);
console.log(birds);
```

CONSOLE

```
["eagle", "condor", "plover", "ostrich", "wren", 46, "penguin", "stork"]
```

The slice() method

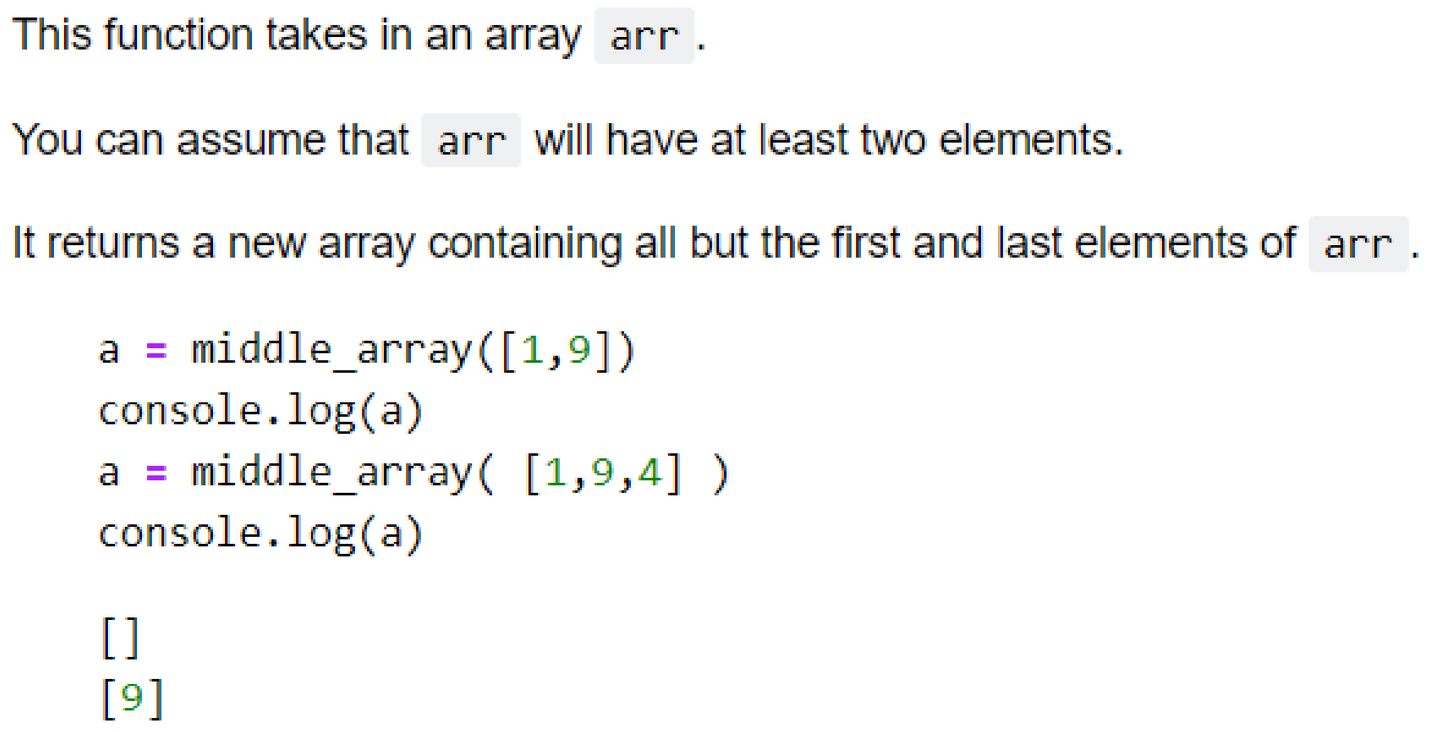
- The **slice(start, [stop])** method is used to slice a specified section of an array and then to create a new array made of the elements indexed from **start** (to **stop** if specified).
- The element indexed by **start** is **included** but the element indexed by **stop** is **excluded**.
- If stop is not specified, the slice() method will slice until the end of the array.



```
let birds = ["eagle", "condor", "plover", "wren", "stork" ];
let somebirds1 = birds.slice(1, 3);
console.log(somebirds1);
let somebirds2 = birds.slice(2);
console.log(somebirds2);
```

```
CONSOLE
["condor", "plover"]
["plover", "wren", "stork"]
```

Activity 2: middle_array(arr)



This function takes in an array `arr`.

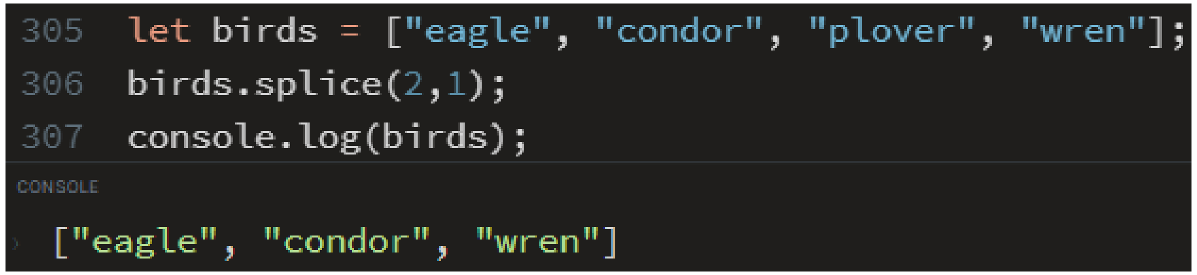
You can assume that `arr` will have at least two elements.

It returns a new array containing all but the first and last elements of `arr`.

```
a = middle_array([1,9])
console.log(a)
a = middle_array( [1,9,4] )
console.log(a)

[]
[9]
```

The splice() method

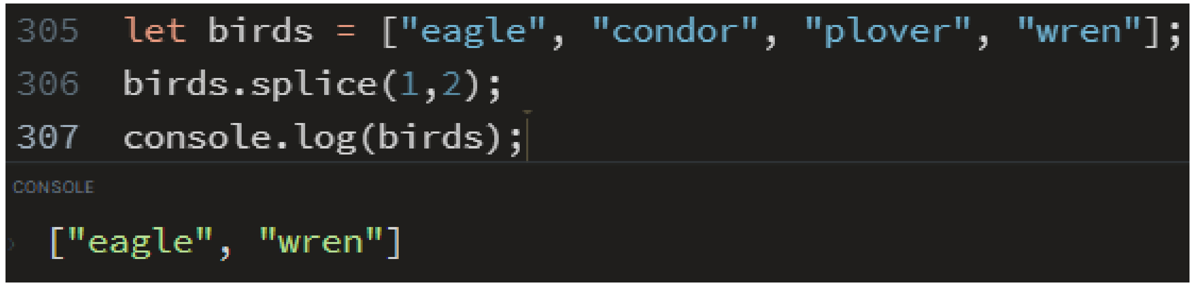
- The **splice(start, num_items)** method is used to remove one or more items.
- E.g., birds.splice(2, 1) will start removing items of list at index 2. The 2nd argument being 1, only 1 item is removed.
- To remove more than one item from **start**, increase the 2nd argument.
- If **num_items** is greater than the remaining number of items after **start**, the rest of the array is removed.



```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(2,1);
console.log(birds);
```

CONSOLE

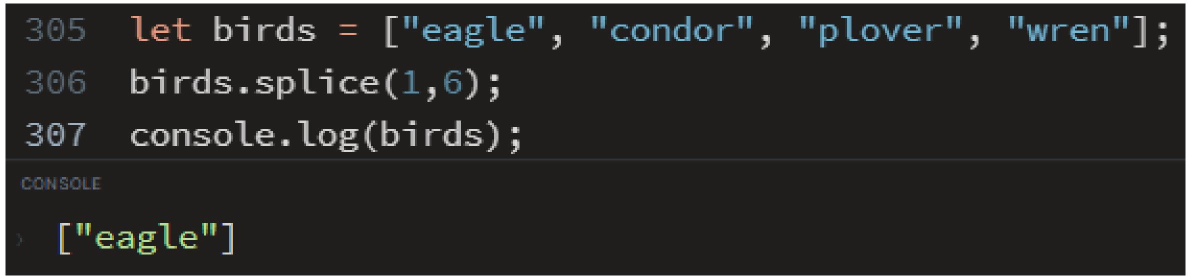
```
["eagle", "condor", "wren"]
```



```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(1,2);
console.log(birds);
```

CONSOLE

```
["eagle", "wren"]
```



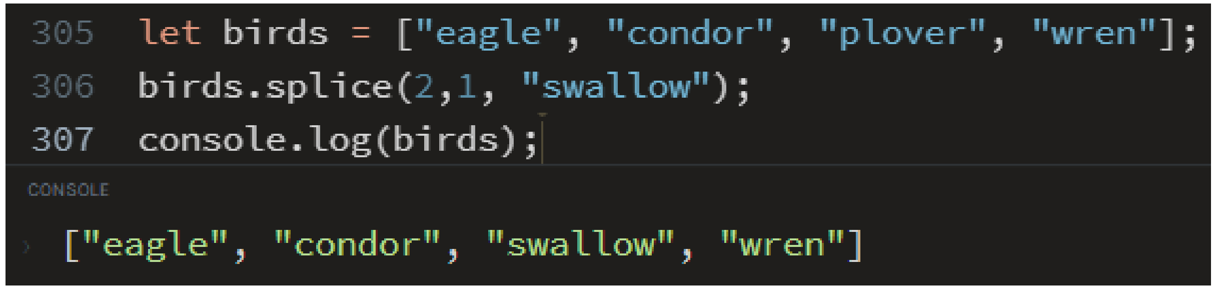
```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(1,6);
console.log(birds);
```

CONSOLE

```
["eagle"]
```

Replacing and inserting items with **splice()**

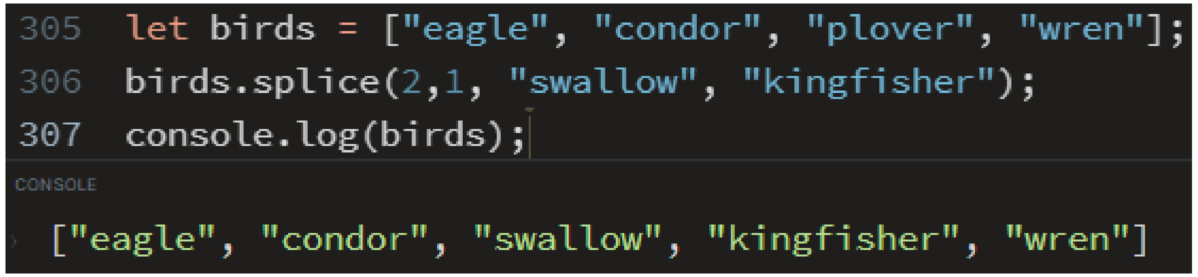
- The **splice(start, num_items, add_items)** method can also be used to replace or insert items.
- E.g., birds.splice(2, 1, “swallow”) will start removing items of list at index 2. The 2nd argument being 1, only 1 item is removed. The 3rd argument replaces the value that was removed.
- You can replace an item by more than one item.
- You can also use this method to insert values in an array by setting the 2nd argument to zero. The insertion starts at the index specified by the 1st argument.



```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(2,1, "swallow");
console.log(birds);
```

CONSOLE

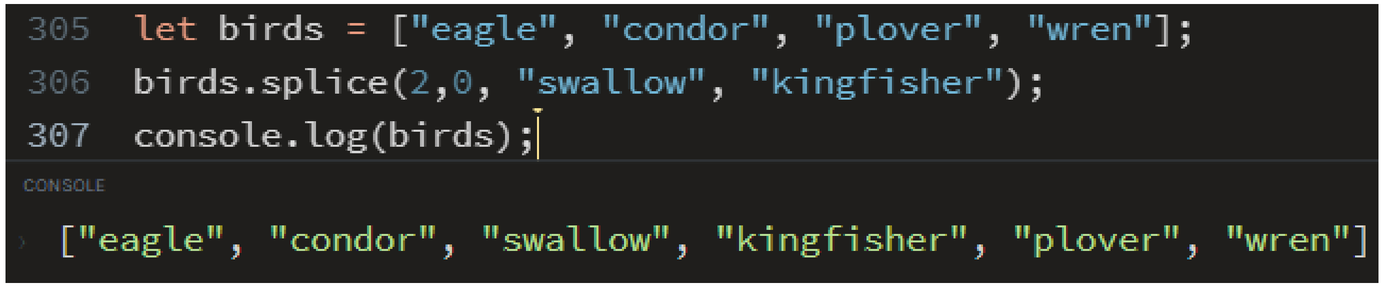
```
["eagle", "condor", "swallow", "wren"]
```



```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(2,1, "swallow", "kingfisher");
console.log(birds);
```

CONSOLE

```
["eagle", "condor", "swallow", "kingfisher", "wren"]
```



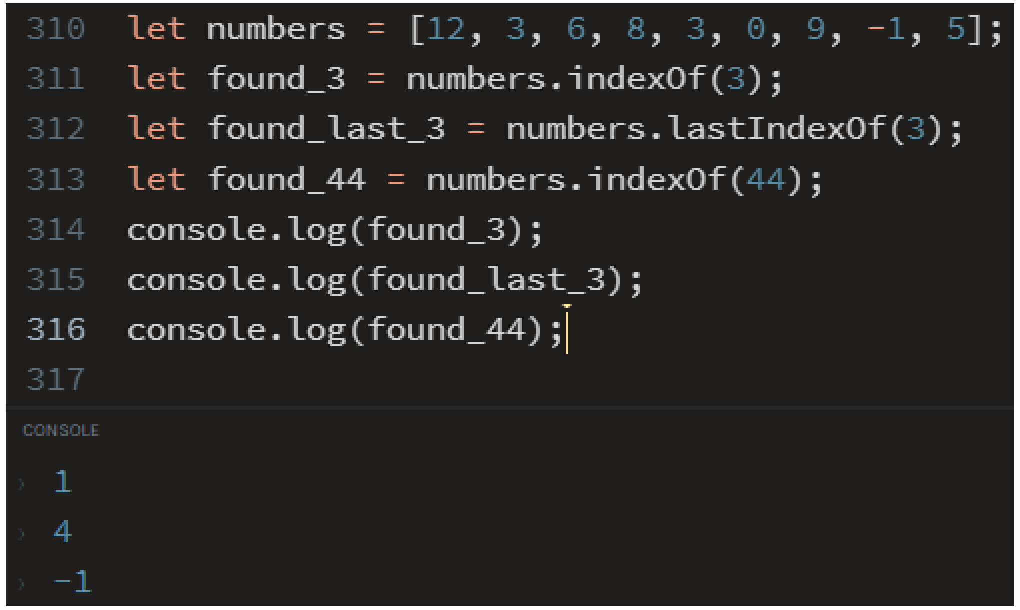
```
let birds = ["eagle", "condor", "plover", "wren"];
birds.splice(2,0, "swallow", "kingfisher");
console.log(birds);
```

CONSOLE

```
["eagle", "condor", "swallow", "kingfisher", "plover", "wren"]
```

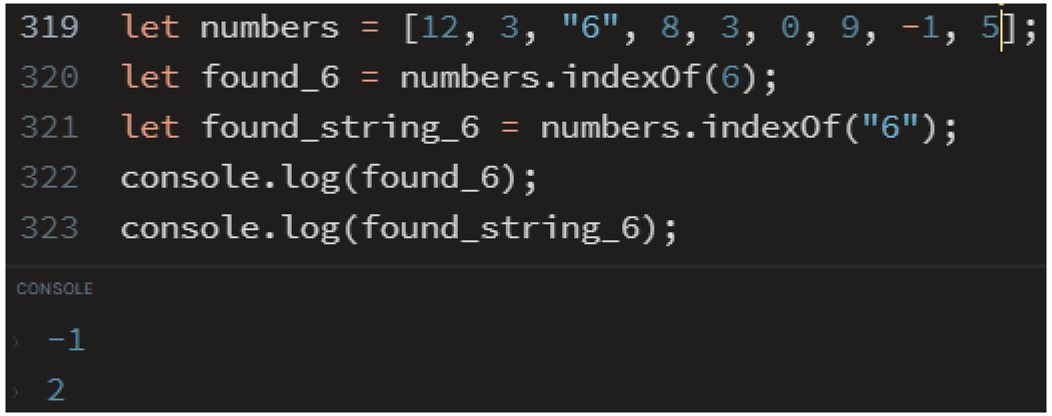
The indexOf() and lastIndexOf() methods

- Used to search for an element in an array.
- indexOf() searches from left to right and lastIndexOf() searches from right to left.
- As soon as the element is found in the array, the methods return its index.
- Otherwise, the methods return -1.
- Both methods are using the === operator instead of the == operator. **No type coercion is performed here.**



```
let numbers = [12, 3, 6, 8, 3, 0, 9, -1, 5];
let found_3 = numbers.indexOf(3);
let found_last_3 = numbers.lastIndexOf(3);
let found_44 = numbers.indexOf(44);
console.log(found_3);
console.log(found_last_3);
console.log(found_44);
```

```
CONSOLE
1
4
-1
```

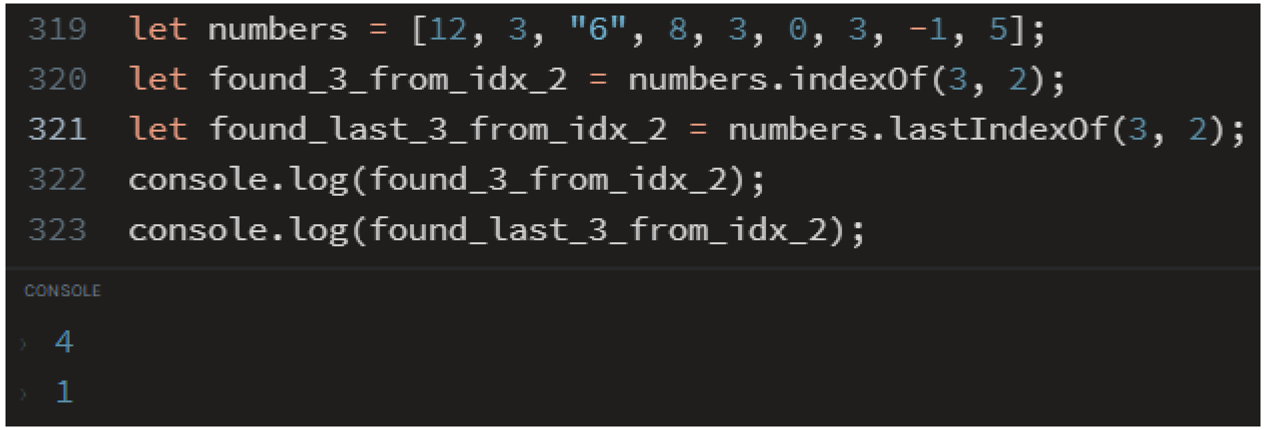


```
let numbers = [12, 3, "6", 8, 3, 0, 9, -1, 5];
let found_6 = numbers.indexOf(6);
let found_string_6 = numbers.indexOf("6");
console.log(found_6);
console.log(found_string_6);
```

```
CONSOLE
-1
2
```

The indexOf() and lastIndexOf() methods

- A 2nd argument can be provided to both methods to start the search from a given index.
- In the examples, the search of value 3 starts from index 2 included, respectively from left to right with indexOf() and from right to left with lastIndexOf().
- Convenient when you need to find all the indices of the array items equal to the element.



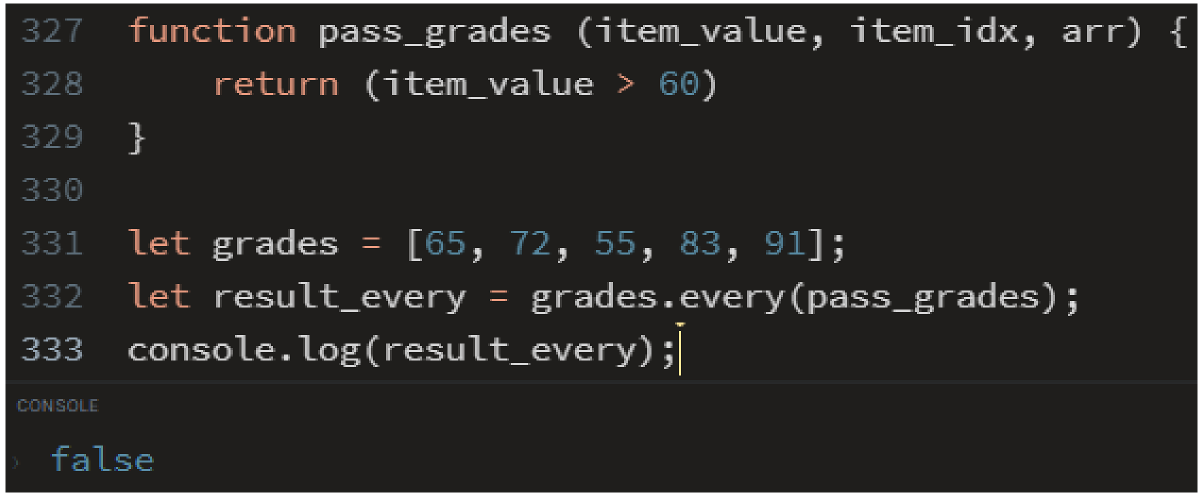
```
let numbers = [12, 3, "6", 8, 3, 0, 3, -1, 5];
let found_3_from_idx_2 = numbers.indexOf(3, 2);
let found_last_3_from_idx_2 = numbers.lastIndexOf(3, 2);
console.log(found_3_from_idx_2);
console.log(found_last_3_from_idx_2);
```

CONSOLE

```
4
1
```

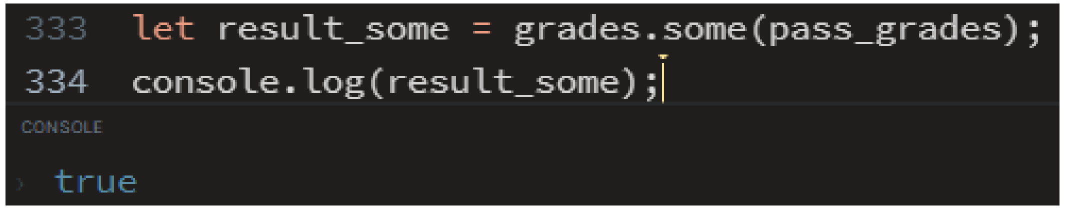
The every(), some() and map() methods

- The argument of these methods is a function returning a Boolean that will be run on each item of the array.
- This function **must receive three arguments**: an item value, an item index and an array name.
- every() returns true if the provided function returns true for all the items of the array.
- some() returns true if the provided function returns true for at least one of the items of the array.
- map() returns an array with the result of each of the function calls on the initial array.



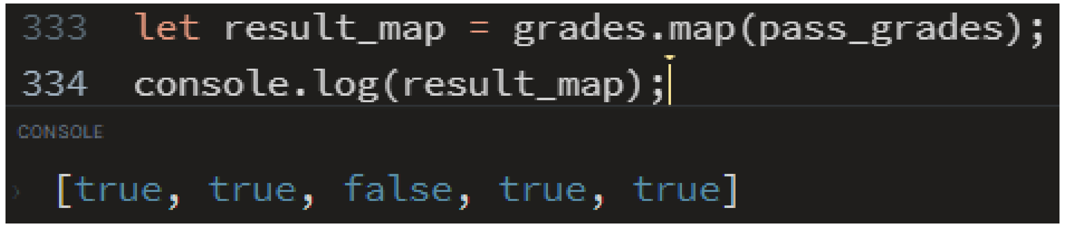
```
function pass_grades (item_value, item_idx, arr) {
    return (item_value > 60)
}

let grades = [65, 72, 55, 83, 91];
let result_every = grades.every(pass_grades);
console.log(result_every);
```

```
false
```



```
let result_some = grades.some(pass_grades);
console.log(result_some);
```

```
true
```

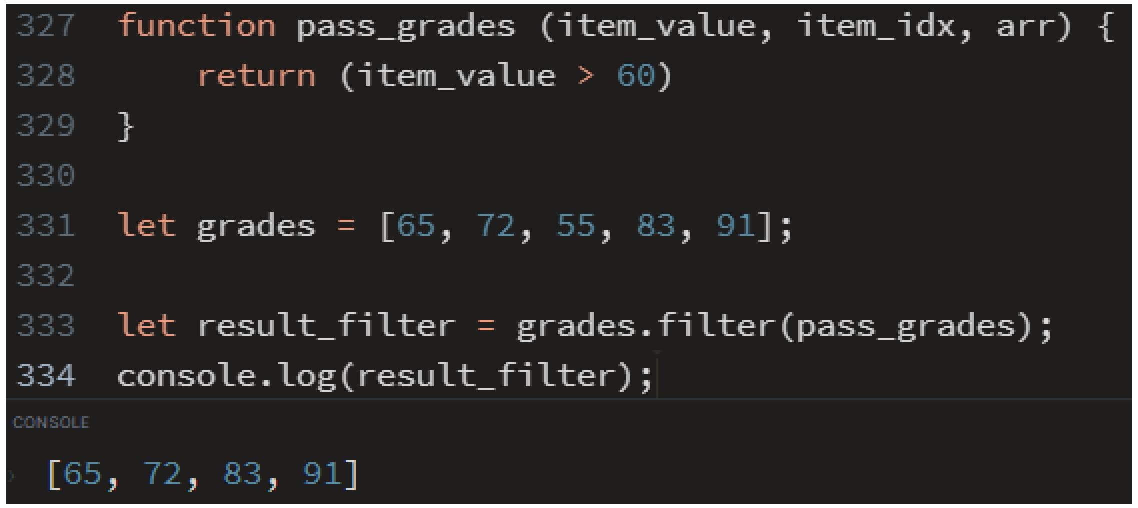


```
let result_map = grades.map(pass_grades);
console.log(result_map);
```

```
[true, true, false, true, true]
```

The filter() and forEach() methods

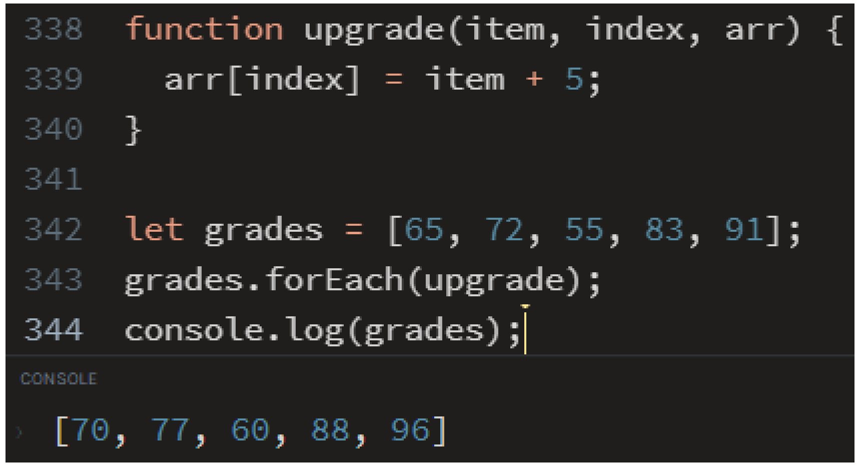
- The argument of these methods is a function that will be run on each item of the array.
- This function **must receive three arguments**: an item value, an item index and an array name.
- filter() executes a provided function (with Booleans returns) for every array item and returns an array of items for which the function returns true.
- forEach() does not return anything but runs the provided function for each item in the array.



```
function pass_grades (item_value, item_idx, arr) {
    return (item_value > 60)
}

let grades = [65, 72, 55, 83, 91];

let result_filter = grades.filter(pass_grades);
console.log(result_filter);
```

CONSOLE

```
[65, 72, 83, 91]
```



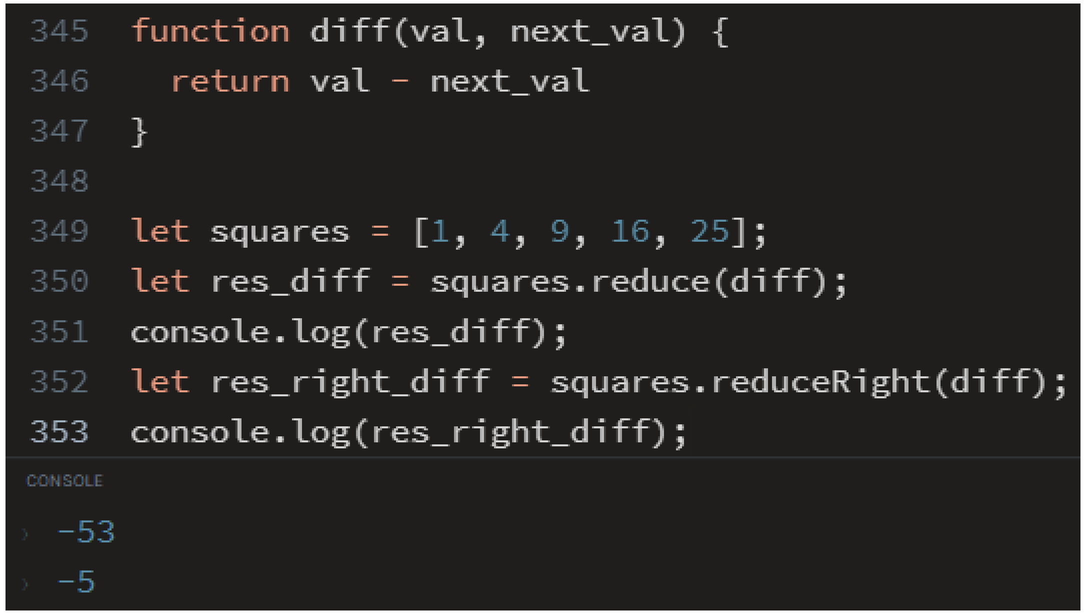
```
function upgrade(item, index, arr) {
  arr[index] = item + 5;
}

let grades = [65, 72, 55, 83, 91];
grades.forEach(upgrade);
console.log(grades);
```

CONSOLE

```
[70, 77, 60, 88, 96]
```

The reduce() and reduceRight() methods

- These methods also iterate over the items in an array, but they build toward a final value that is returned.
- Argument: a function that receives itself two arguments, the accumulated value and the current value.
- reduce() takes the array items from left to right and executes the diff function. So, initially, 1 and 4 are used.
- Their difference, -3, is calculated and will be passed as the first argument of the diff function at the next iteration. The second argument will be the next item in the array (so, 9).
- The returned value is the last difference (-28 - 25 = -53)



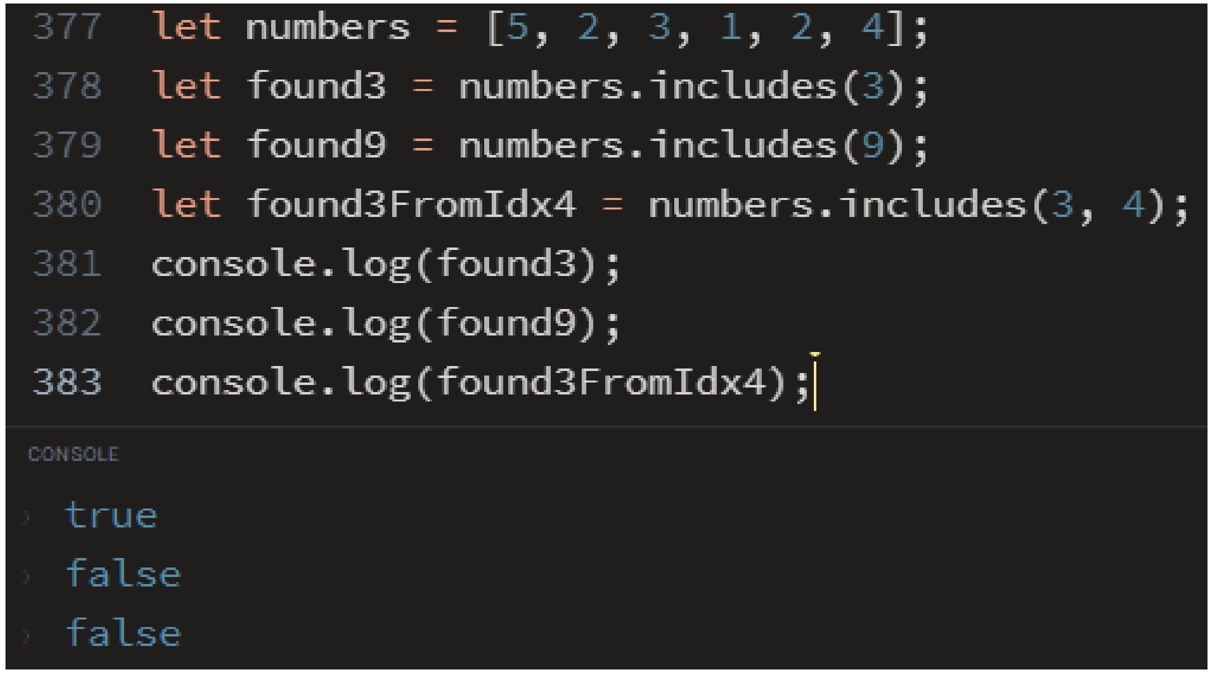
```
function diff(val, next_val) {
  return val - next_val
}

let squares = [1, 4, 9, 16, 25];
let res_diff = squares.reduce(diff);
console.log(res_diff);
let res_right_diff = squares.reduceRight(diff);
console.log(res_right_diff);
```

```
CONSOLE
-53
-5
```

- reduceRight() is similar to reduce(), except that it takes the array items from right to left. The first iteration will take 25 as first and 16 as second parameters.

The includes() method

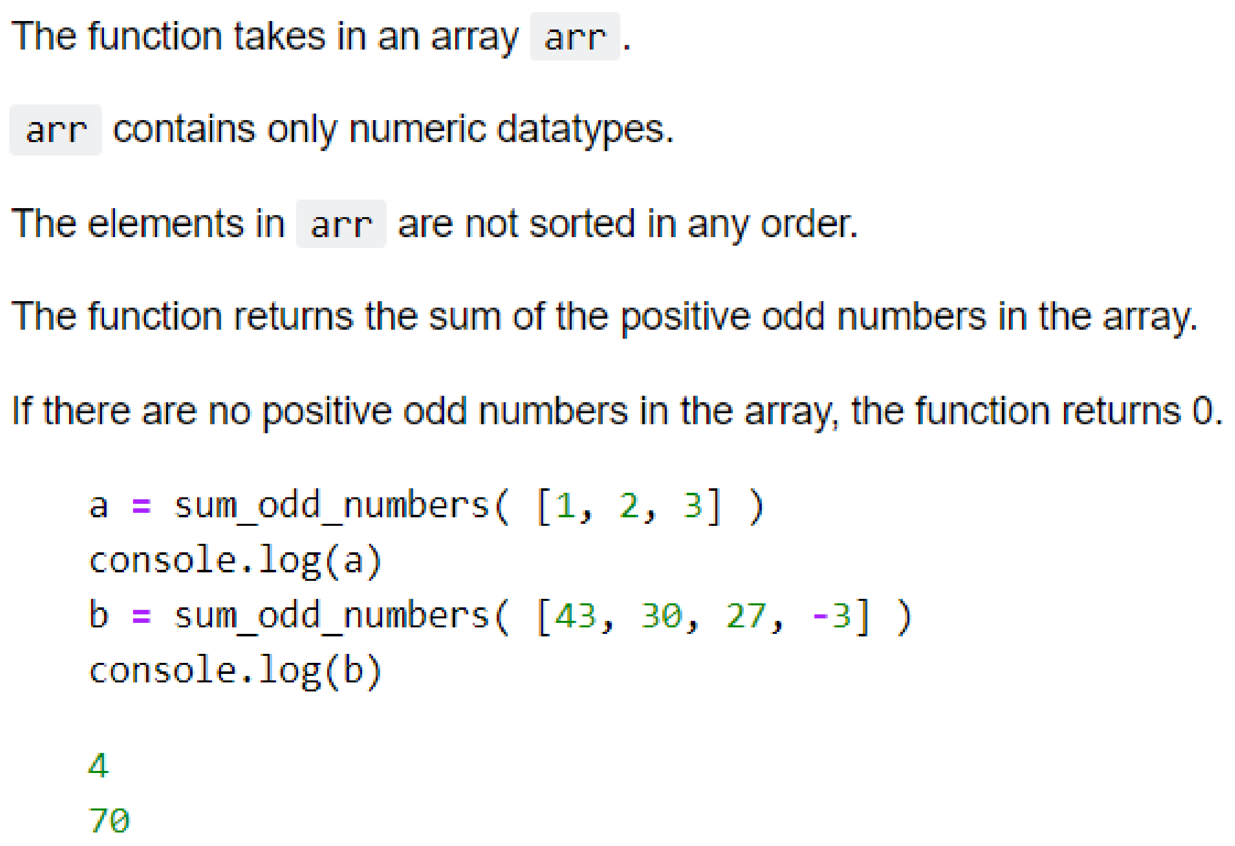
- includes() is used to check whether an element belongs to the array.
- Unlike indexOf(), lastIndexOf(), find() and findIndex(), it returns a Boolean: true if the element is in the array, false otherwise.
- You can also provide a starting index from which the search begins.



```javascript
let numbers = [5, 2, 3, 1, 2, 4];
let found3 = numbers.includes(3);
let found9 = numbers.includes(9);
let found3FromIdx4 = numbers.includes(3, 4);
console.log(found3);
console.log(found9);
console.log(found3FromIdx4);
```

```
true
false
false
```

Activity 3: sum_odd_numbers(arr)



The function takes in an array `arr`.

`arr` contains only numeric datatypes.

The elements in `arr` are not sorted in any order.

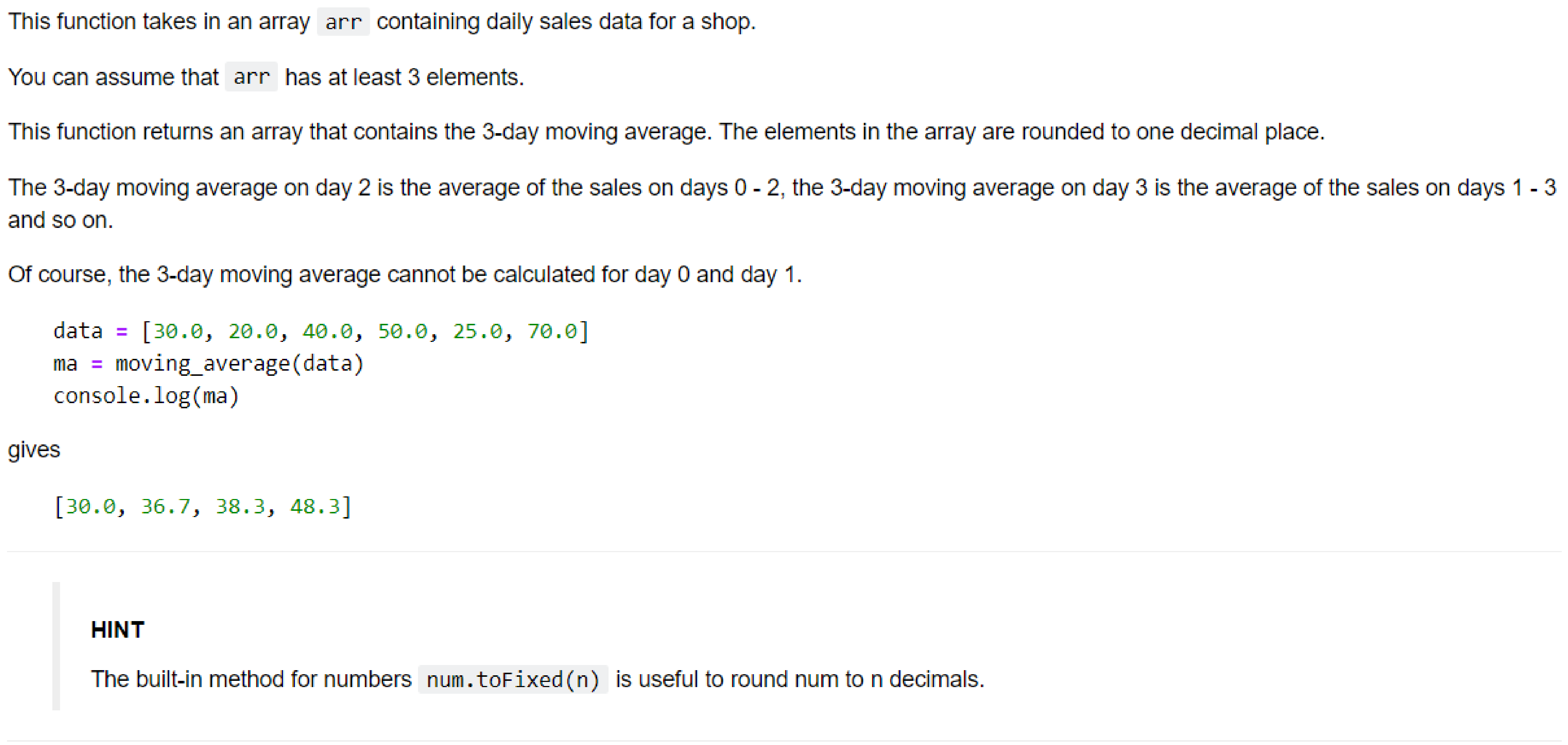
The function returns the sum of the positive odd numbers in the array.

If there are no positive odd numbers in the array, the function returns 0.

```
a = sum_odd_numbers( [1, 2, 3] )
console.log(a)
b = sum_odd_numbers( [43, 30, 27, -3] )
console.log(b)
```

```
4
70
```

Activity 4: moving_average(arr)



This function takes in an array `arr` containing daily sales data for a shop.

You can assume that `arr` has at least 3 elements.

This function returns an array that contains the 3-day moving average. The elements in the array are rounded to one decimal place.

The 3-day moving average on day 2 is the average of the sales on days 0 - 2, the 3-day moving average on day 3 is the average of the sales on days 1 - 3 and so on.

Of course, the 3-day moving average cannot be calculated for day 0 and day 1.

```
data = [30.0, 20.0, 40.0, 50.0, 25.0, 70.0]
ma = moving_average(data)
console.log(ma)
```

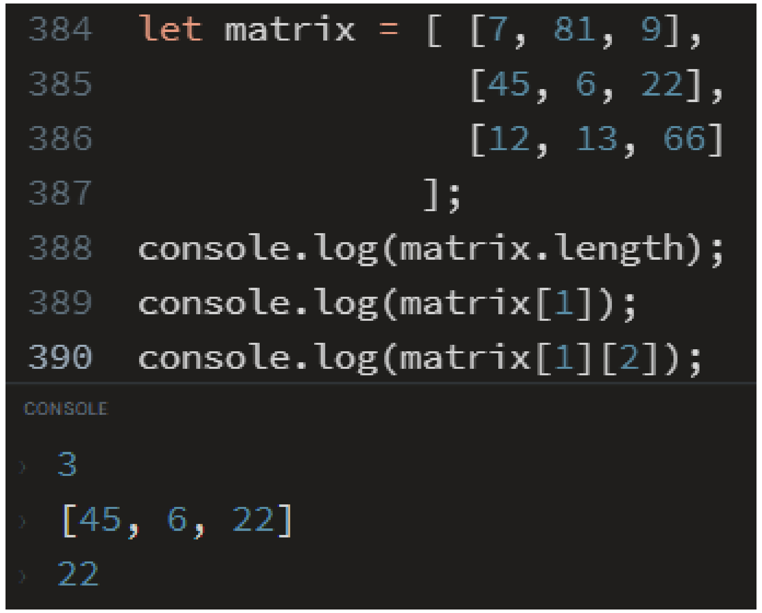
gives

```
[30.0, 36.7, 38.3, 48.3]
```

> **HINT**
>
> The built-in method for numbers `num.toFixed(n)` is useful to round num to n decimals.

Nested Arrays

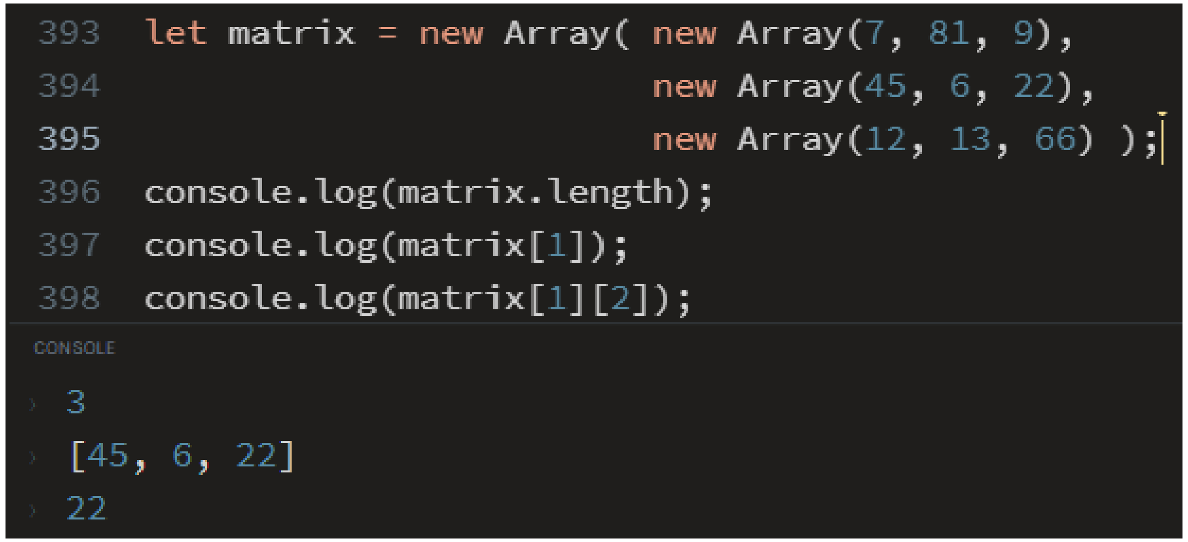
- The items of an array are not limited to basic types but can also be complex objects. In particular, arrays can be nested.
- Arrays of arrays are standard structures to describe tables, matrices, lists of lists.
- They can be declared as before, using nested square bracket literals or nested new operators. The latter is not recommended.
- Note that the variable **matrix** only contains 3 items, each of them being an array of 3 numbers.
- To access the i-th row of matrix, with i >= 0: **matrix[i]**.
- To access the j-th element of the i-th row of matrix, with both i, j >= 0: **matrix[i][j]**.



```
let matrix = [ [7, 81, 9],
               [45, 6, 22],
               [12, 13, 66]
             ];
console.log(matrix.length);
console.log(matrix[1]);
console.log(matrix[1][2]);
```

CONSOLE

```
3
[45, 6, 22]
22
```



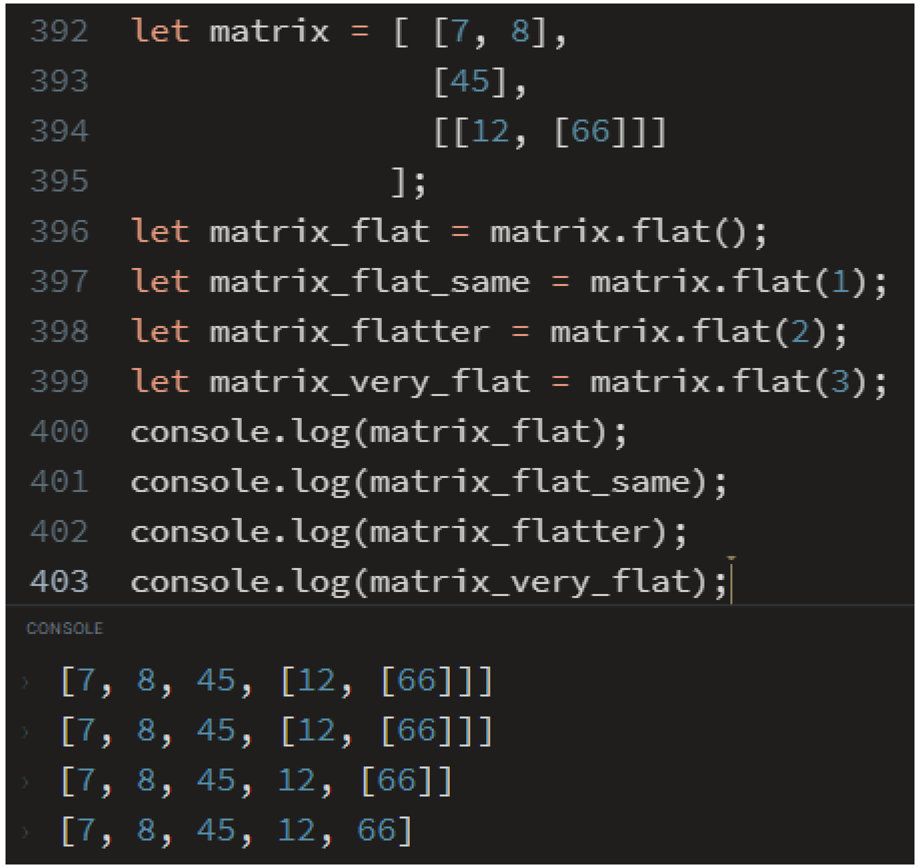
```
let matrix = new Array( new Array(7, 81, 9),
                        new Array(45, 6, 22),
                        new Array(12, 13, 66) );
console.log(matrix.length);
console.log(matrix[1]);
console.log(matrix[1][2]);
```

CONSOLE

```
3
[45, 6, 22]
22
```

The flat() method

- The flat() method is used to flatten nested arrays into an array.
- Concretely, it removes one pair of square brackets around each item of an array, if they had one.
- flat() can take a number argument n, which is equivalent to apply the flat() method n times over the array.
- In the example, flat() or flat(1) flattens [7, 8], and [45]. But [[12, [66]]] needs to be flattened 3 times for all the brackets to be removed.



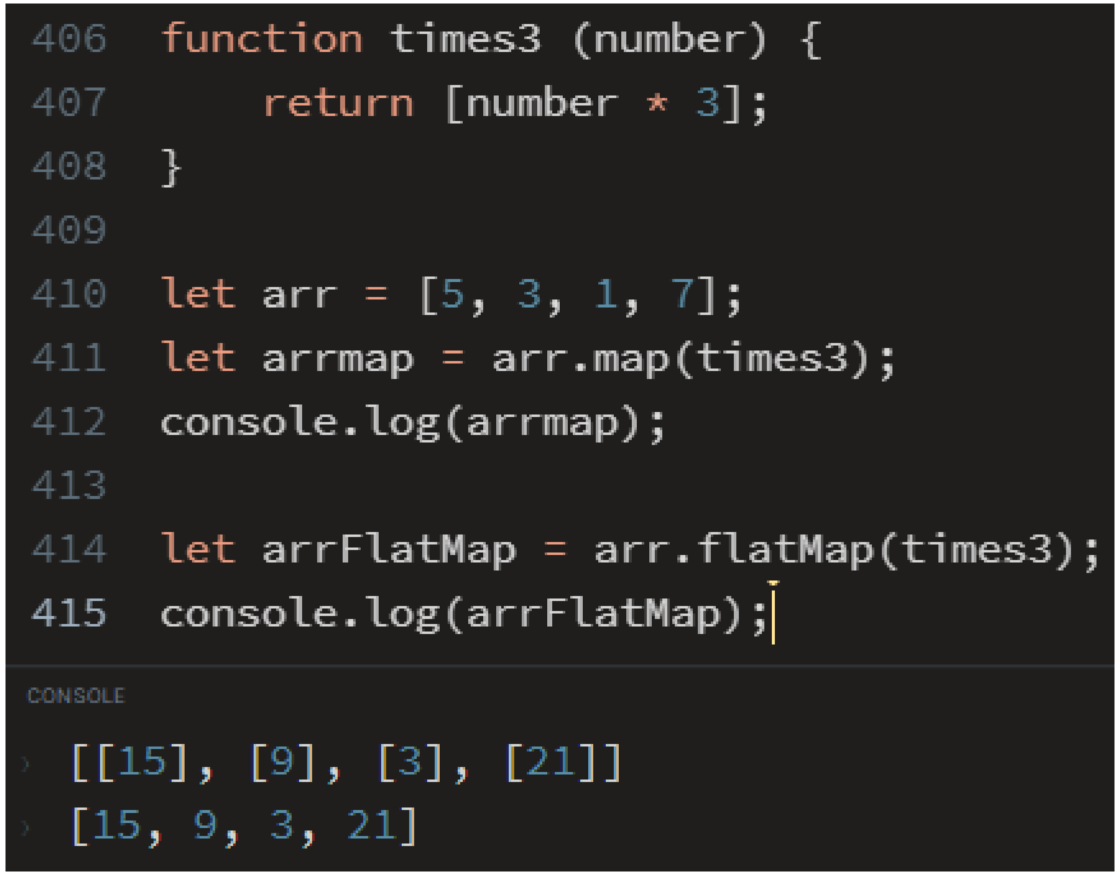
```
let matrix = [ [7, 8],
               [45],
               [[12, [66]]]
             ];
let matrix_flat = matrix.flat();
let matrix_flat_same = matrix.flat(1);
let matrix_flatter = matrix.flat(2);
let matrix_very_flat = matrix.flat(3);
console.log(matrix_flat);
console.log(matrix_flat_same);
console.log(matrix_flatter);
console.log(matrix_very_flat);
```

CONSOLE

```
[7, 8, 45, [12, [66]]]
[7, 8, 45, [12, [66]]]
[7, 8, 45, 12, [66]]
[7, 8, 45, 12, 66]
```

The flatMap() method

- The flatMap() is a combination of the map() and flat(1) methods.
- Concretely, it first apply the map() method over the array, given a predefined function.
- Then, it applies the flat() method.

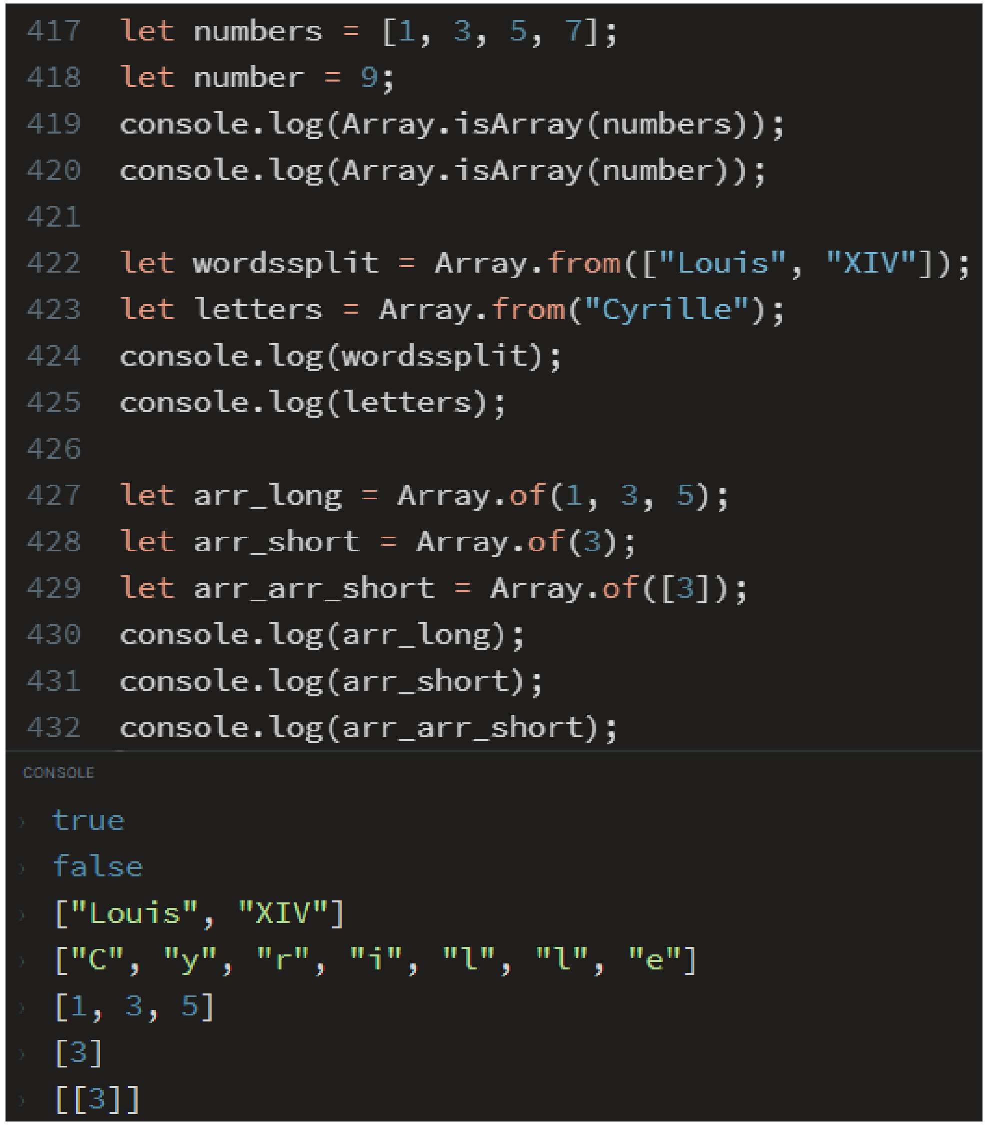


```
function times3 (number) {
    return [number * 3];
}

let arr = [5, 3, 1, 7];
let arrmap = arr.map(times3);
console.log(arrmap);

let arrFlatMap = arr.flatMap(times3);
console.log(arrFlatMap);
```

```
CONSOLE
[[15], [9], [3], [21]]
[15, 9, 3, 21]
```

The Array.isArray/from/of() methods

- The Array.name_method() are called using the dot notation over the Array keyword because the items they operate on might not be an array at the time these methods receive them.
- E.g., Array.isArray(arg) checks whether arg is an array or not.
- Array.from() creates an array from various types of data.
- Array.of() creates an array containing the values sent to it as arguments.
- Note that new Array(3) and Array.of(3) are different, the 1st returning an empty array of length 3 whereas the 2nd returns an array of length 1 containing the value 3.

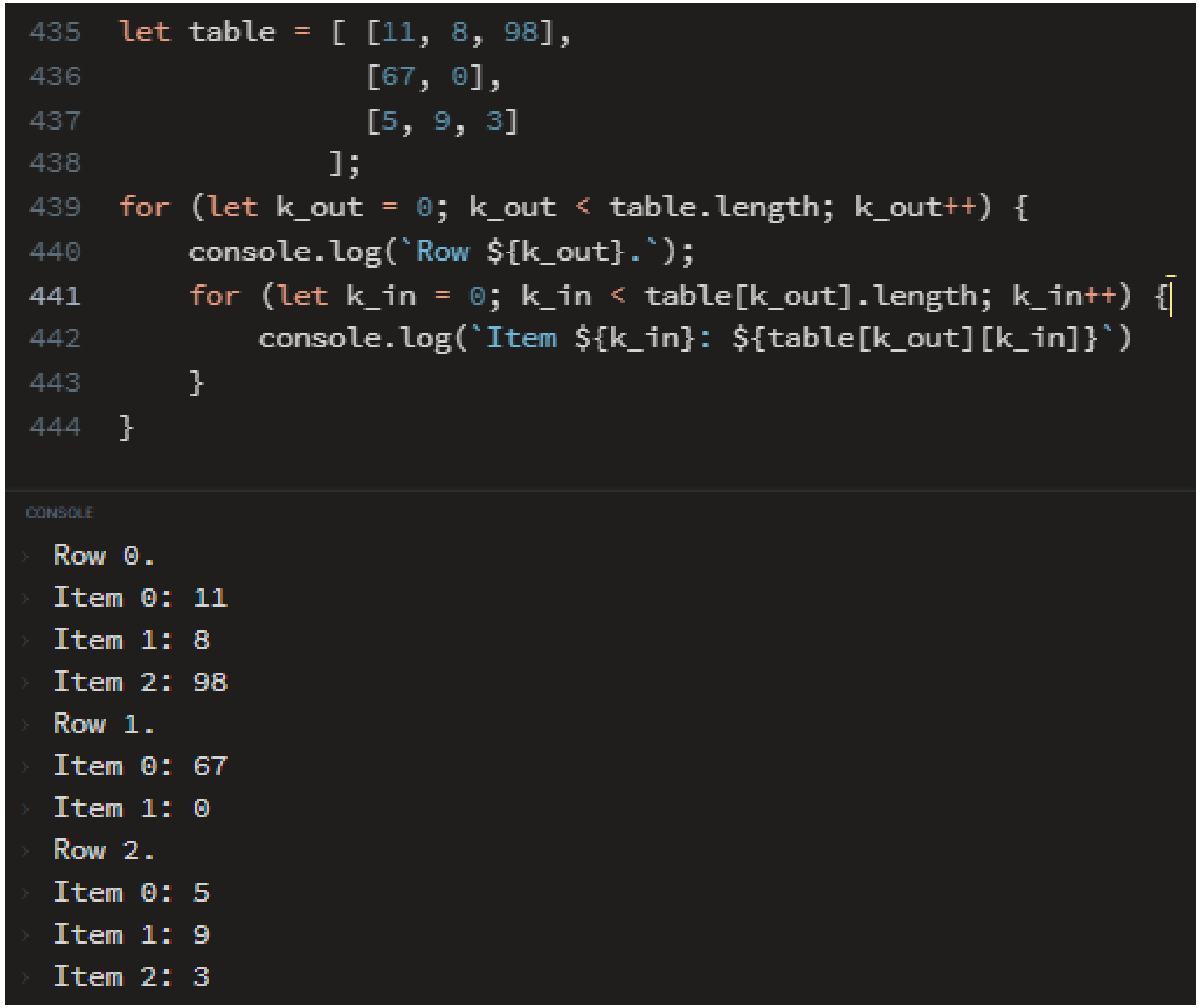


```
let numbers = [1, 3, 5, 7];
let number = 9;
console.log(Array.isArray(numbers));
console.log(Array.isArray(number));

let wordssplit = Array.from(["Louis", "XIV"]);
let letters = Array.from("Cyrille");
console.log(wordssplit);
console.log(letters);

let arr_long = Array.of(1, 3, 5);
let arr_short = Array.of(3);
let arr_arr_short = Array.of([3]);
console.log(arr_long);
console.log(arr_short);
console.log(arr_arr_short);
```

```
CONSOLE
true
false
["Louis", "XIV"]
["C", "y", "r", "i", "l", "l", "e"]
[1, 3, 5]
[3]
[[3]]
```

Traversing nested arrays

- To loop through nested arrays, you need to create a nested loop.
- The outer loop iterates over the items of the array (e.g., over the rows of a table). The loop variable varies from 0 to the number of items or rows (using the table.length property).
- The inner loop, for each sub-array, iterates over the elements of the current sub-array. The loop variable varies from 0 to the number of elements in current row k (using the table[k].length property).



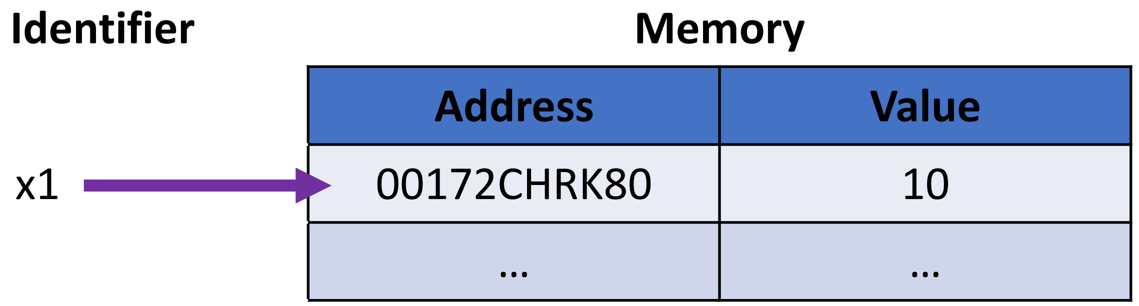
```
let table = [ [11, 8, 98],
              [67, 0],
              [5, 9, 3]
            ];
for (let k_out = 0; k_out < table.length; k_out++) {
    console.log(`Row ${k_out}.`);
    for (let k_in = 0; k_in < table[k_out].length; k_in++) {
        console.log(`Item ${k_in}: ${table[k_out][k_in]}`)
    }
}
```

CONSOLE

```
Row 0.
Item 0: 11
Item 1: 8
Item 2: 98
Row 1.
Item 0: 67
Item 1: 0
Row 2.
Item 0: 5
Item 1: 9
Item 2: 3
```

Memory of a computer

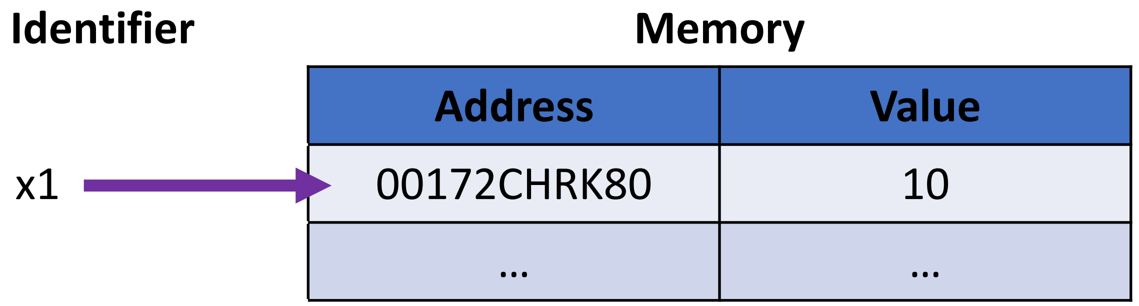
- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by a **value**, which corresponds to the **address** of the “box”.



Address	Value
00172CHRK80	10
…	…

Memory of a computer

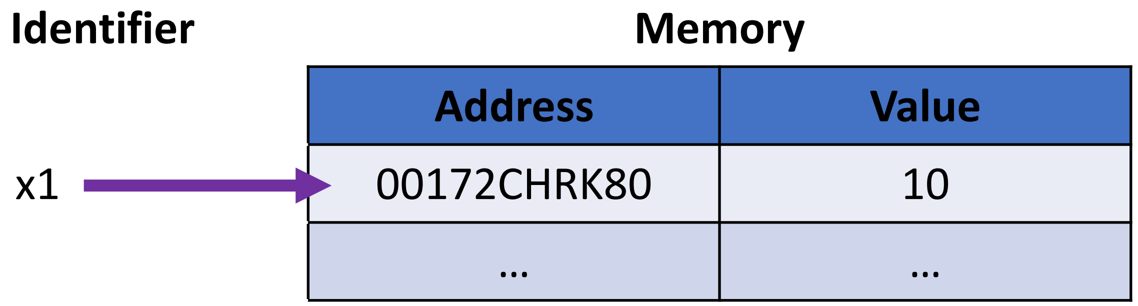
- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by a **value**, which corresponds to the **address** of the “box”.
- When a variable is created:
  - A “box” is assigned for the variable and its value is stored in the “box”.
  - The variable name simply refers to the address of the “box”.



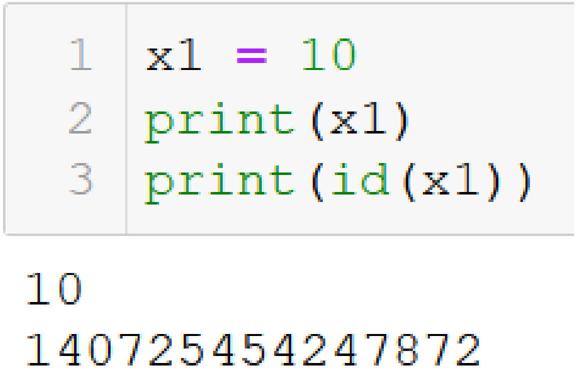
Address	Value
00172CHRK80	10
…	…

Memory of a computer

- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by a **value**, which corresponds to the **address** of the “box”.
- When a variable is created:
  - A “box” is assigned for the variable and its value is stored in the “box”.
  - The variable name simply refers to the address or the ID of the “box”.



Address	Value
00172CHRK80	10
…	…



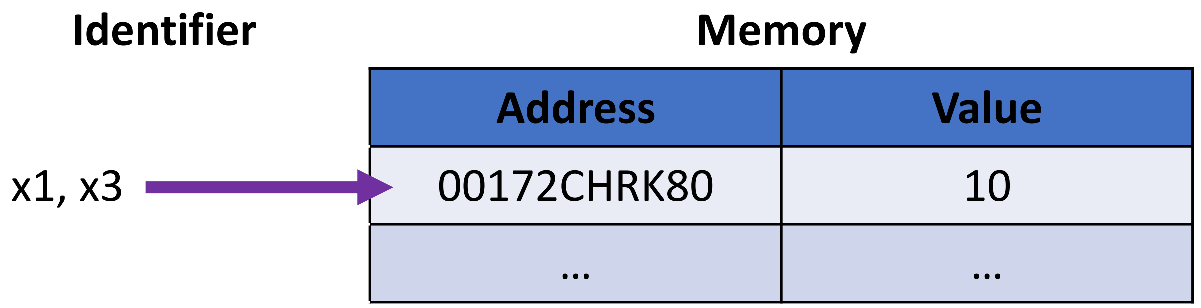
```python
x1 = 10
print(x1)
print(id(x1))
```

```
10
140725454247872
```

Example in Python.
In JavaScript, there is no id() function

Aliasing

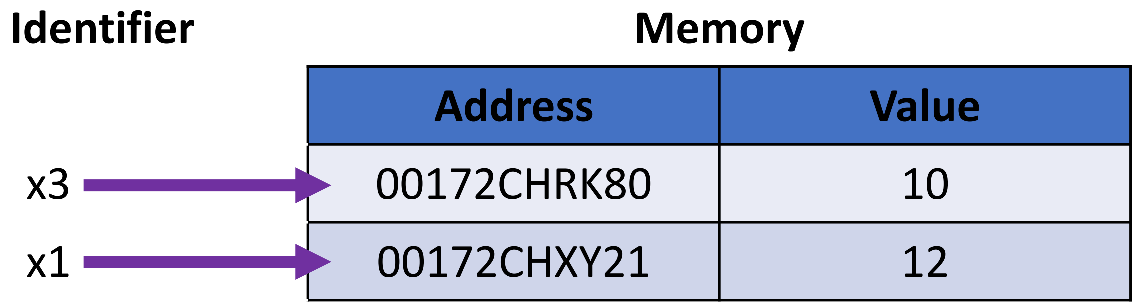
- **Aliasing:** JavaScript saves memory space by having two variable names **point to the same memory address**.
- Two variable names with identical values will have the same address in memory.



Address	Value
00172CHRK80	10
…	…

Aliasing

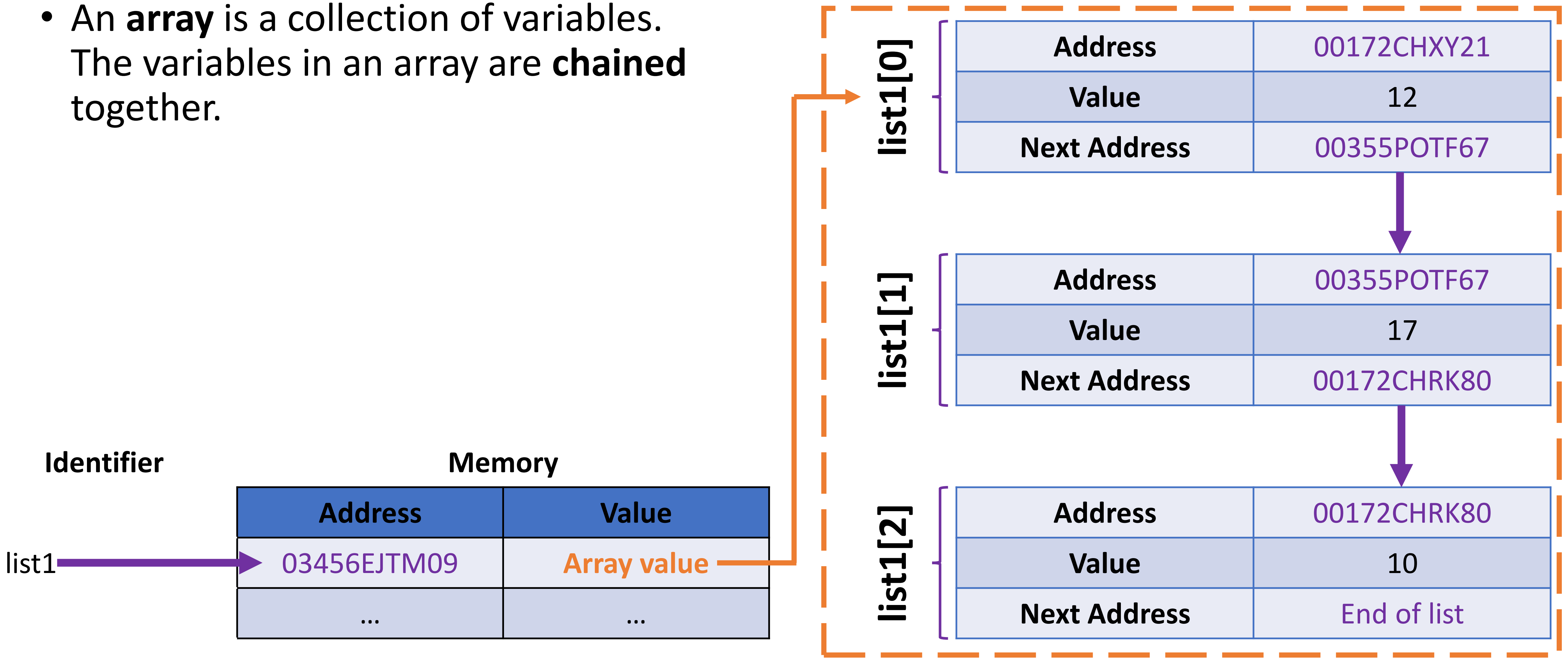
- **Aliasing:** JavaScript saves memory space by having two variable names **point to the same memory address**.
- Two variable names with identical values will have the same address in memory.
- Otherwise, the variable names will point to different memory address.



Address	Value
00172CHRK80	10
00172CHXY21	12

Memory management in arrays

- An **array** is a collection of variables. The variables in an array are **chained** together.



Address	Value
03456EJTM09	Array value
…	…

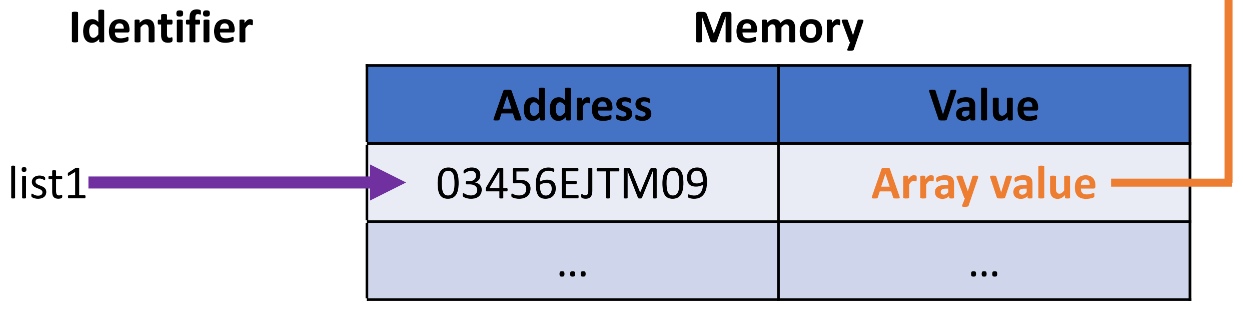
Address	00172CHXY21
Value	12
Next Address	00355POTF67

Address	00355POTF67
Value	17
Next Address	00172CHRK80

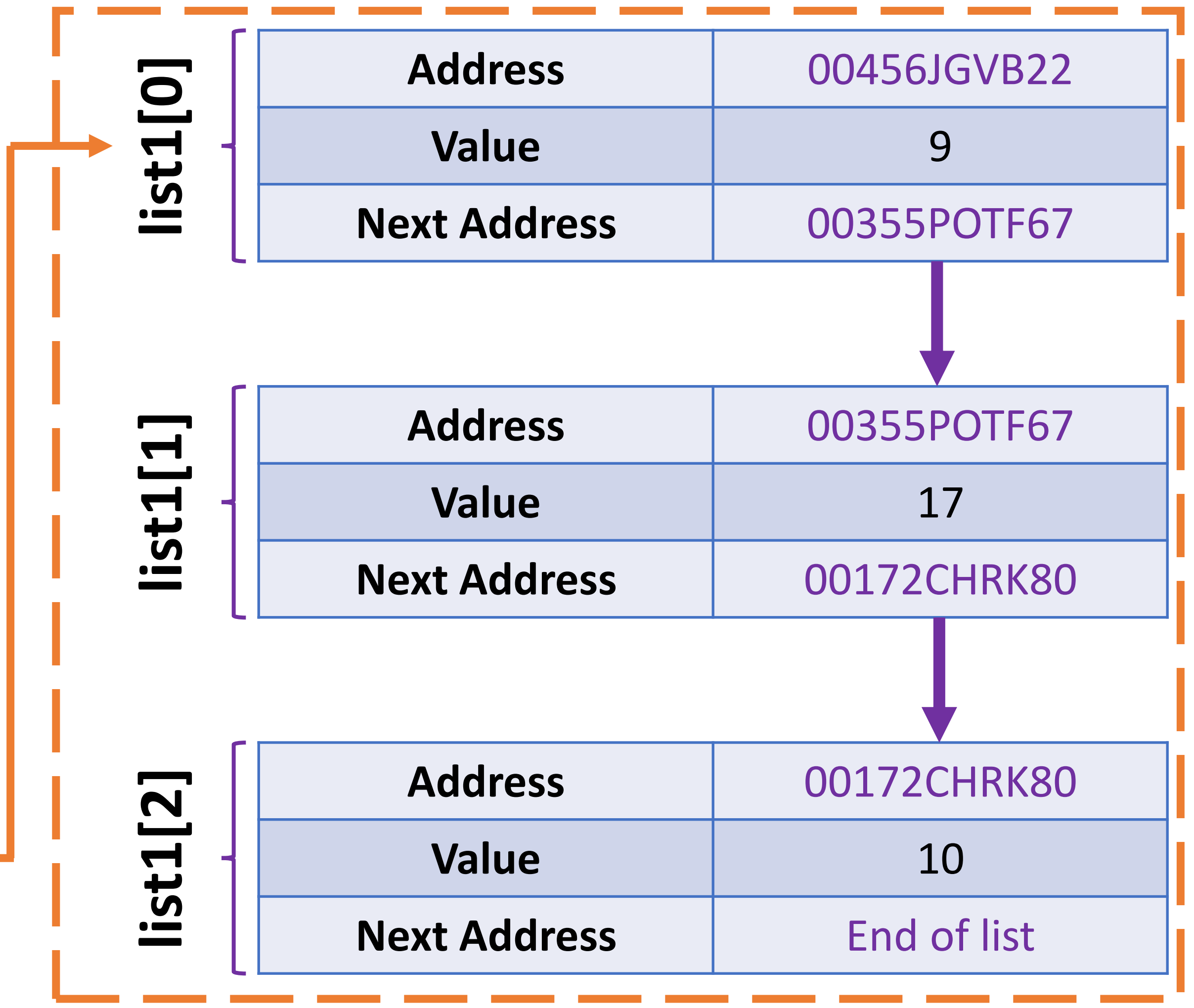
Address	00172CHRK80
Value	10
Next Address	End of list

Memory management in arrays

- An **array** is a collection of variables. The variables in an array are **chained** together.
- If x1 is changed, JS will adjust so that the array remains unaffected. It simply reallocates x1 to another location in memory.
- However, if list1[0] is changed, the address of list1[0] is updated but not the address of the array.



Address	Value
03456EJTM09	Array value
…	…



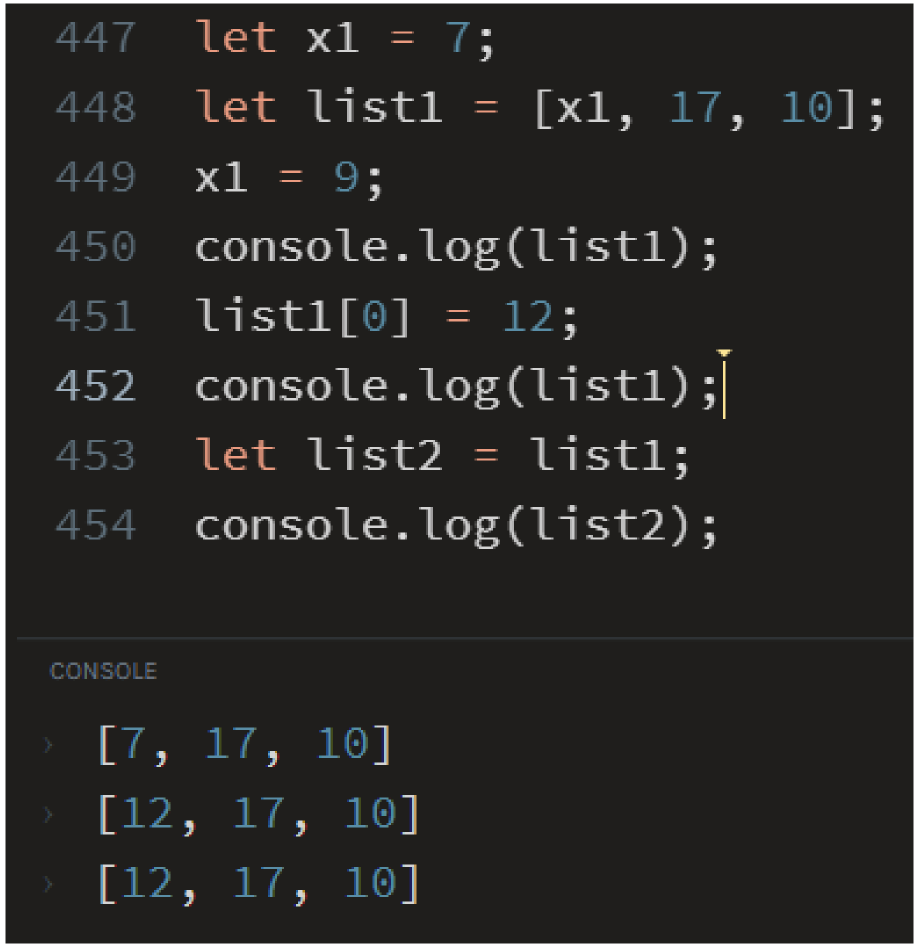
Address	00456JGVB22
Value	9
Next Address	00355POTF67

Address	00355POTF67
Value	17
Next Address	00172CHRK80

Address	00172CHRK80
Value	10
Next Address	End of list

Aliasing in arrays: problem

- An **array** is a collection of variables. The variables in an array are **chained** together.
- **Aliasing:** We can assign an array to another variable name.



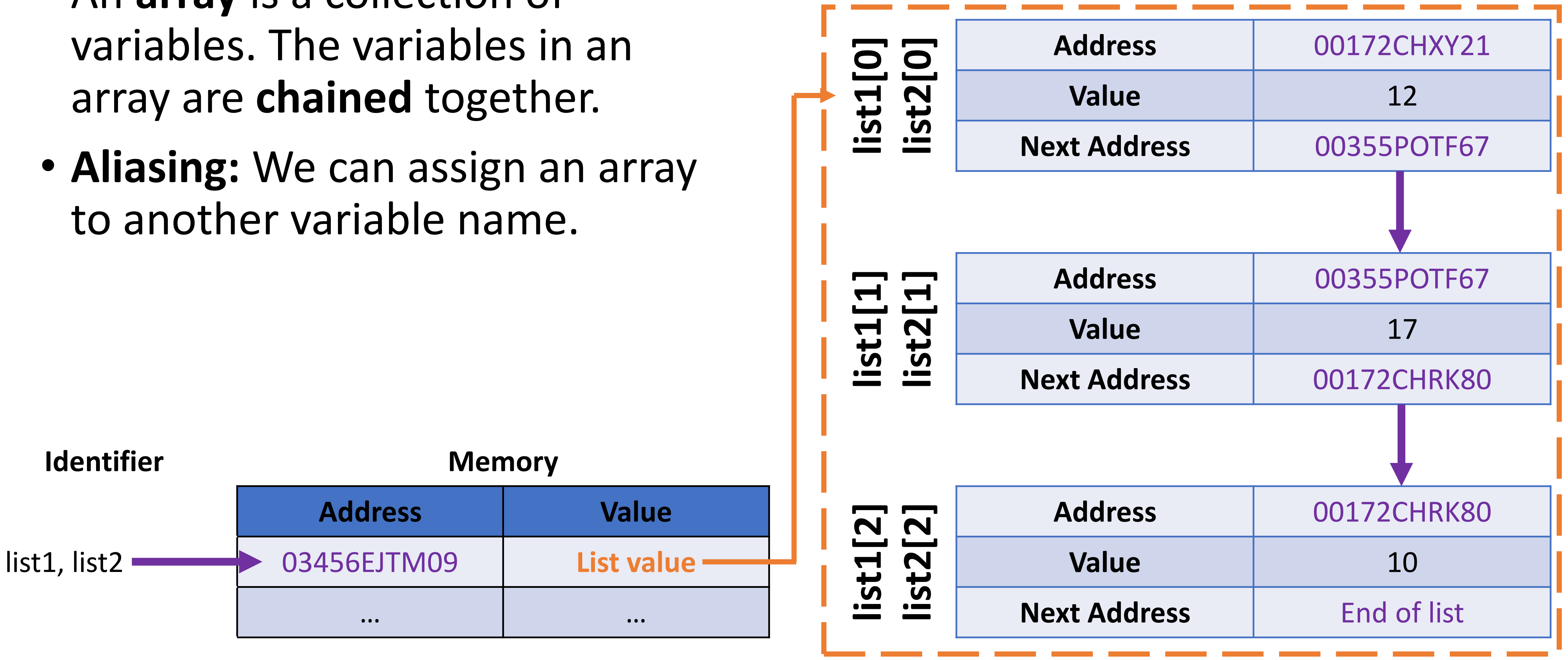
```
let x1 = 7;
let list1 = [x1, 17, 10];
x1 = 9;
console.log(list1);
list1[0] = 12;
console.log(list1);
let list2 = list1;
console.log(list2);
```

CONSOLE

```
[7, 17, 10]
[12, 17, 10]
[12, 17, 10]
```

Aliasing in arrays: problem

- An **array** is a collection of variables. The variables in an array are **chained** together.
- **Aliasing:** We can assign an array to another variable name.



Identifier — **Memory**

Identifier	Address	Value
list1, list2	03456EJTM09	List value
	…	…

list1[0] list2[0]

Address	00172CHXY21
Value	12
Next Address	00355POTF67

list1[1] list2[1]

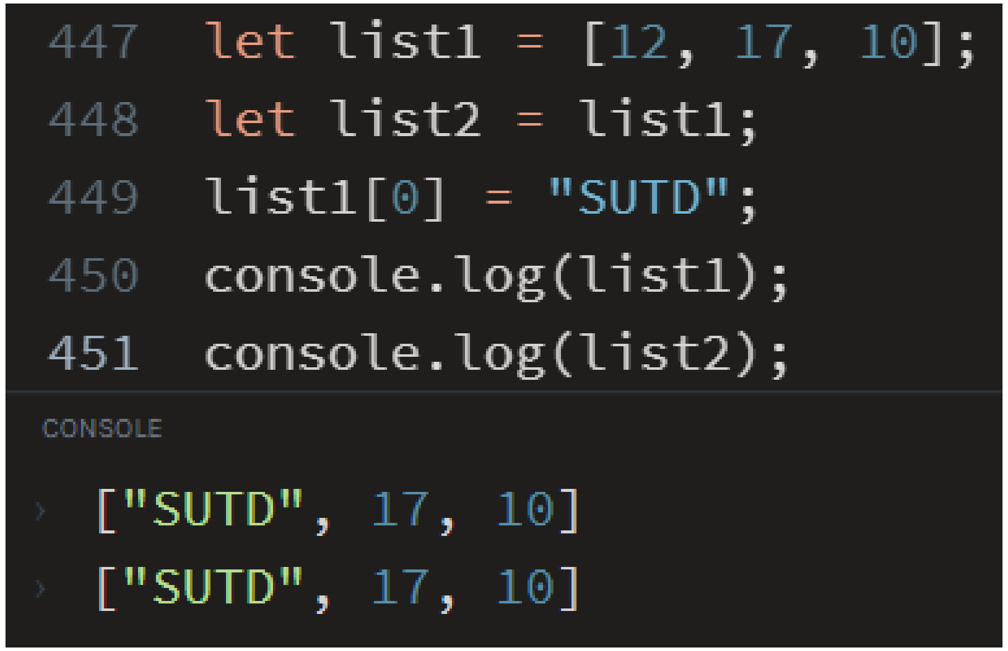
Address	00355POTF67
Value	17
Next Address	00172CHRK80

list1[2] list2[2]

Address	00172CHRK80
Value	10
Next Address	End of list

Aliasing in arrays: problem

- An **array** is a collection of variables. The variables in an array are **chained** together.
- **Aliasing:** We can assign an array to another variable name.
- **Problem:** changing list1[0] changes list1 values, but also changes list2.



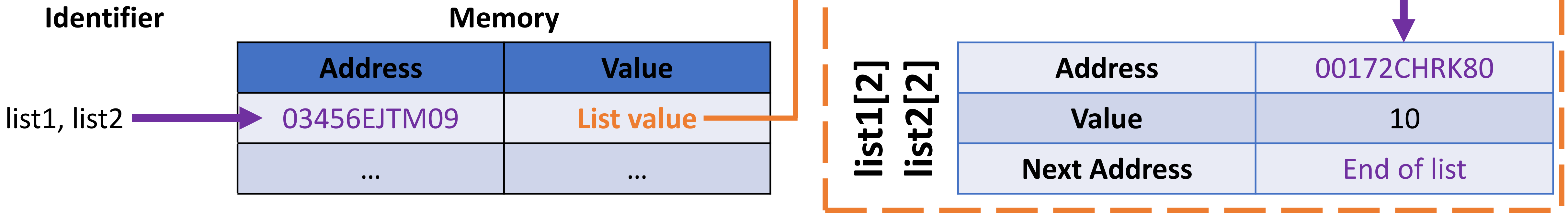
```
let list1 = [12, 17, 10];
let list2 = list1;
list1[0] = "SUTD";
console.log(list1);
console.log(list2);
```

CONSOLE

```
› ["SUTD", 17, 10]
› ["SUTD", 17, 10]
```

Aliasing in arrays: problem

- An **array** is a collection of variables. The variables in an array are **chained** together.
- **Aliasing:** We can assign an array to another variable name.
- **Problem:** changing list1[0] changes list1 values, but also changes list2.



Address	Value
03456EJTM09	List value
…	…

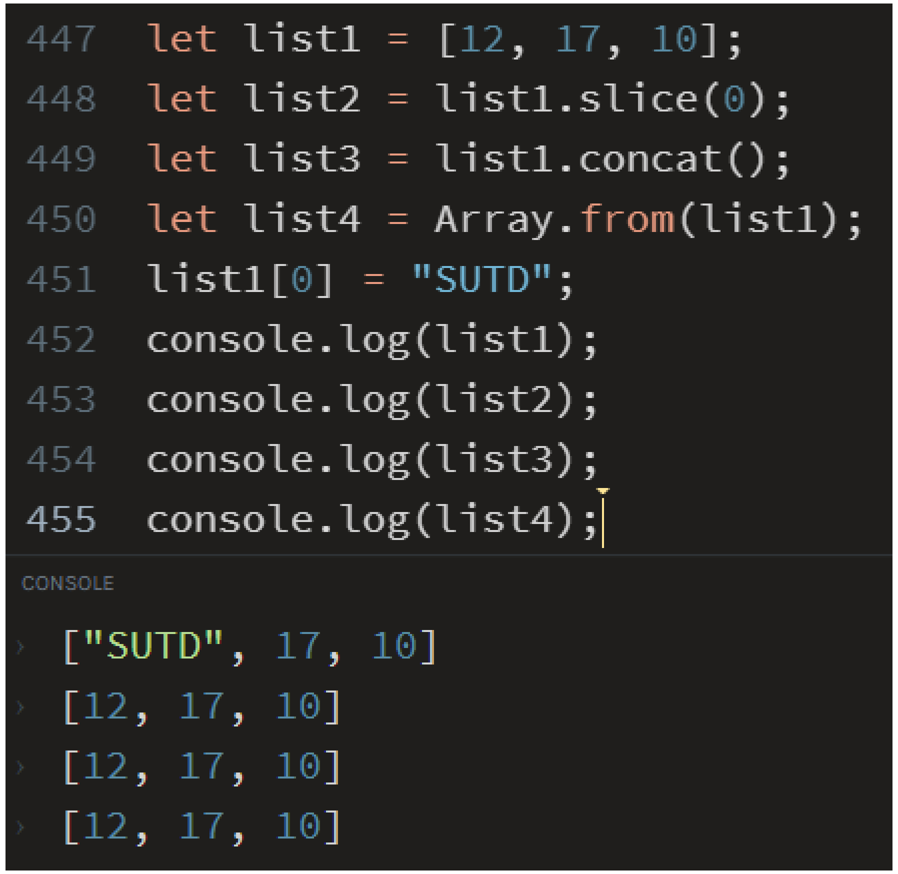
Address	00172CHXY21
Value	12 → "SUTD"
Next Address	00355POTF67

Address	00355POTF67
Value	17
Next Address	00172CHRK80

Address	00172CHRK80
Value	10
Next Address	End of list

Shallow copy of an array

- **Problem:** changing list1[0] changes list1 values, but also changes list2.
- **Shallow copy:** list1.slice(0) makes list2 a shallow copy of list1. By doing so, list2 will be saved to its own location of memory.
- Changing a value in list1, with list1[index] = …, no longer affects the shallow copies.
- Note: There exist other techniques to make a shallow copy. E.g.,
    - list3 = list1.concat()
    - List4 = Array.from(list1)



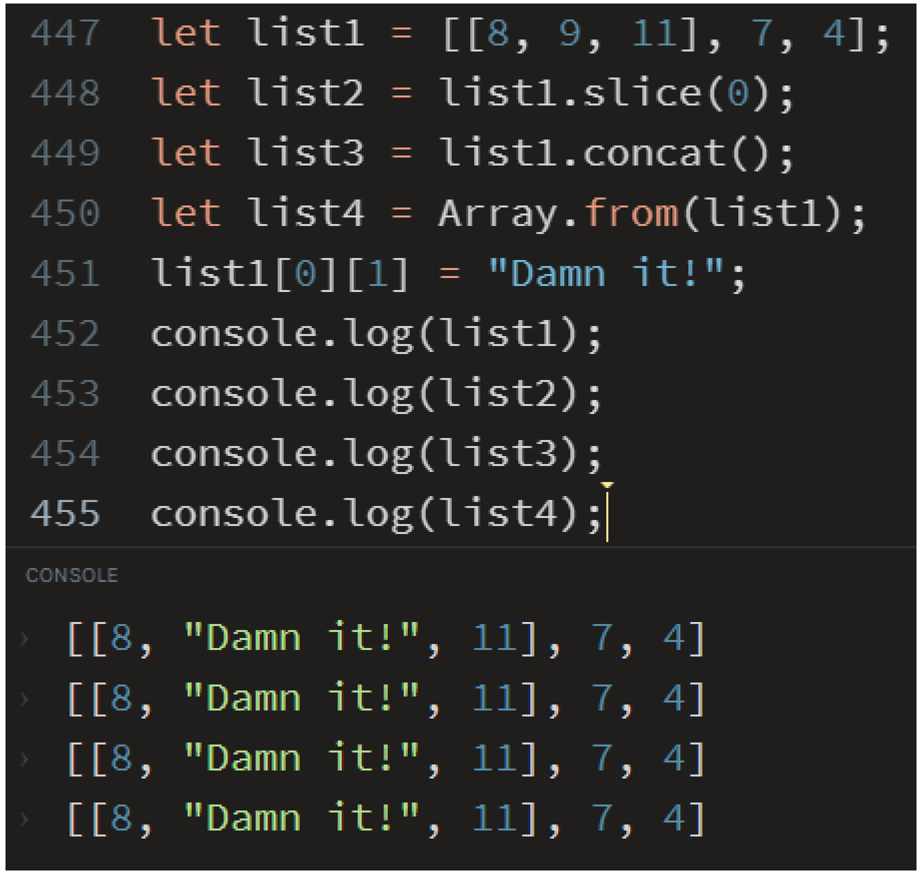
```
let list1 = [12, 17, 10];
let list2 = list1.slice(0);
let list3 = list1.concat();
let list4 = Array.from(list1);
list1[0] = "SUTD";
console.log(list1);
console.log(list2);
console.log(list3);
console.log(list4);
```

CONSOLE

```
["SUTD", 17, 10]
[12, 17, 10]
[12, 17, 10]
[12, 17, 10]
```

Shallow copy: problem

- **Note:** if an element of an array is an array (e.g., matrices), then the shallow copy will not copy the sub-arrays to different locations of memory.
- **Problem:** changing a sub-array element then affects both arrays, even though these arrays are shallow copies of each other.



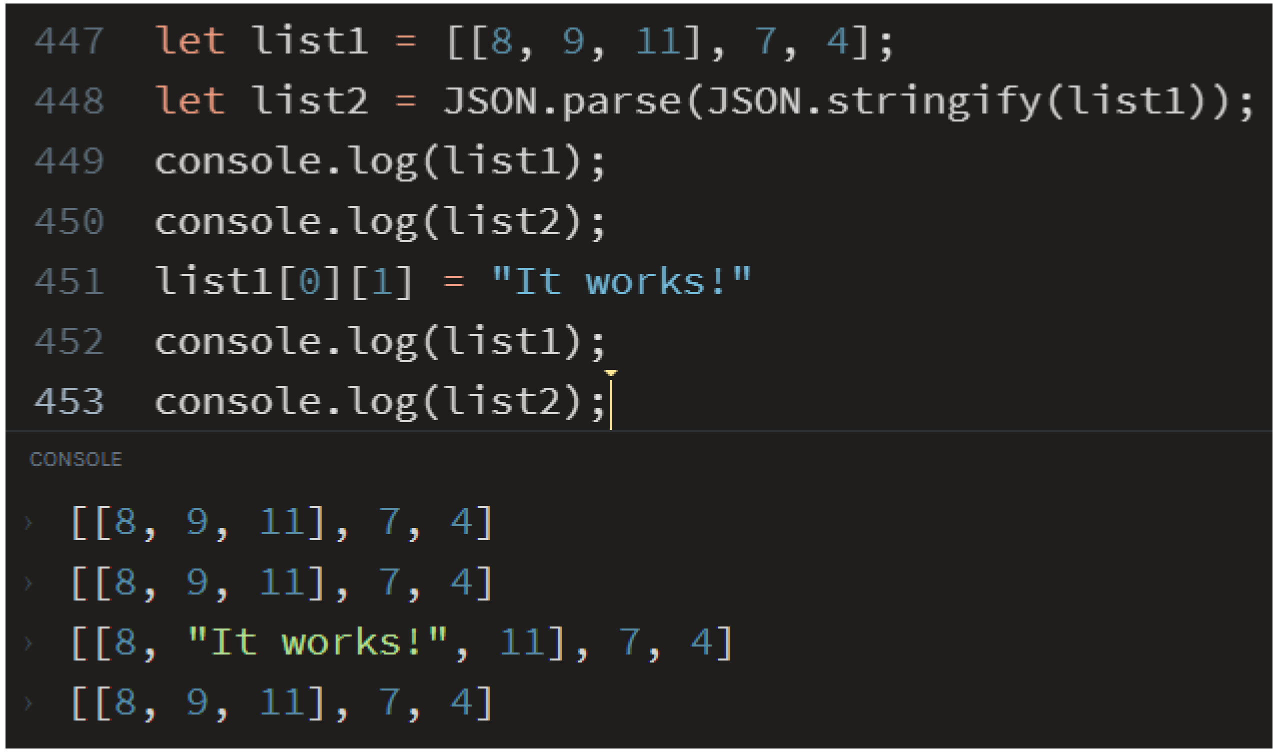
```
let list1 = [[8, 9, 11], 7, 4];
let list2 = list1.slice(0);
let list3 = list1.concat();
let list4 = Array.from(list1);
list1[0][1] = "Damn it!";
console.log(list1);
console.log(list2);
console.log(list3);
console.log(list4);
```

CONSOLE

```
[[8, "Damn it!", 11], 7, 4]
[[8, "Damn it!", 11], 7, 4]
[[8, "Damn it!", 11], 7, 4]
[[8, "Damn it!", 11], 7, 4]
```

Deep copy

- **Solution:** make a **deep copy.**
- A deep copy forces JS to make sure **all elements and sub-elements** are assigned to different locations in memory.
- Making a deep copy is not trivial and requires JS to use advanced functions and objects:
- JSON.parse(JSON.stringify(list1)) will create a deep copy of list1, fully independent of the original array.
- More on JSON later!



```javascript
let list1 = [[8, 9, 11], 7, 4];
let list2 = JSON.parse(JSON.stringify(list1));
console.log(list1);
console.log(list2);
list1[0][1] = "It works!"
console.log(list1);
console.log(list2);
```

CONSOLE

```
[[8, 9, 11], 7, 4]
[[8, 9, 11], 7, 4]
[[8, "It works!", 11], 7, 4]
[[8, 9, 11], 7, 4]
```

To summarize…

- To copy a **single value** x1, just do an **aliasing**: x2 = x1. Although they share the same address at this stage, both variables are "independent". Modifying one of them does not affect the other.
- To copy an "independent" **(simple) array** l2 of l1, a **shallow copy** is fine: l2 = l1.slice(0) or l2 = l1.concat() or l2 = Array.from(l1).
- But if **list l1 contains at least one sublist**, a **deep copy** is necessary:
  - l2 = JSON.parse(JSON.stringify(list1))

Great advice

Great Advice #: Keep the aliasing, shallow and deep copies concepts in mind for now.

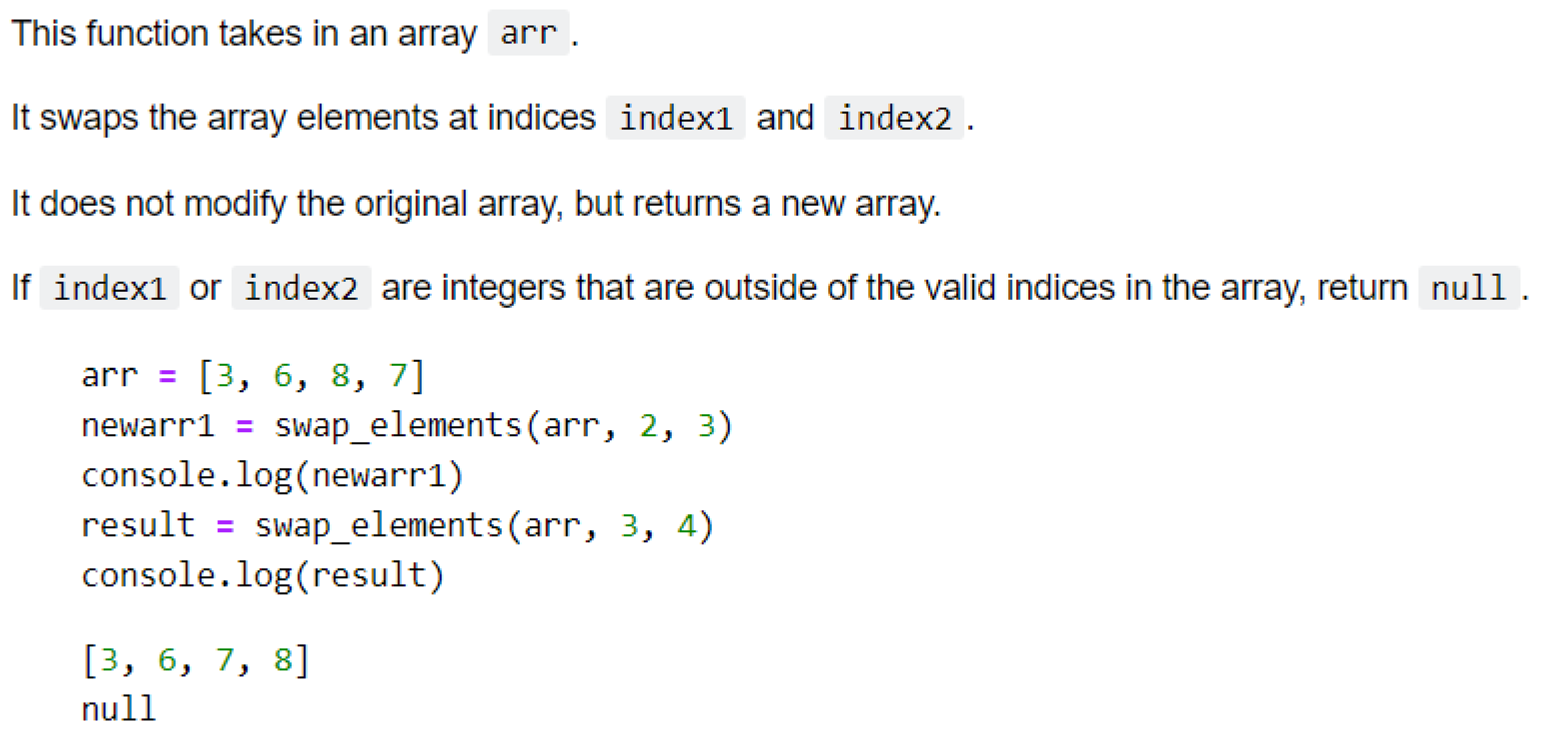
If you find that modifying an array object ends up unexpectedly changing another, then you might have an aliasing or shallow copy problem.

When in doubt, make a deep copy.

Do not worry about understanding all these memory concepts, these are definitely not trivial!

However, **try not to have doubts** and don’t do deep copies when not necessary! ☺

Activity 5 : swap_elements(arr, index1, index2)



This function takes in an array `arr`.

It swaps the array elements at indices `index1` and `index2`.

It does not modify the original array, but returns a new array.

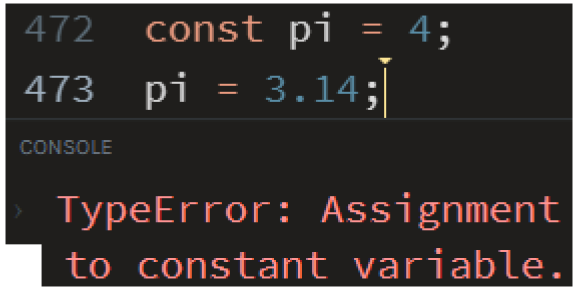
If `index1` or `index2` are integers that are outside of the valid indices in the array, return `null`.

```
arr = [3, 6, 8, 7]
newarr1 = swap_elements(arr, 2, 3)
console.log(newarr1)
result = swap_elements(arr, 3, 4)
console.log(result)
```

```
[3, 6, 7, 8]
null
```

Let or const for arrays

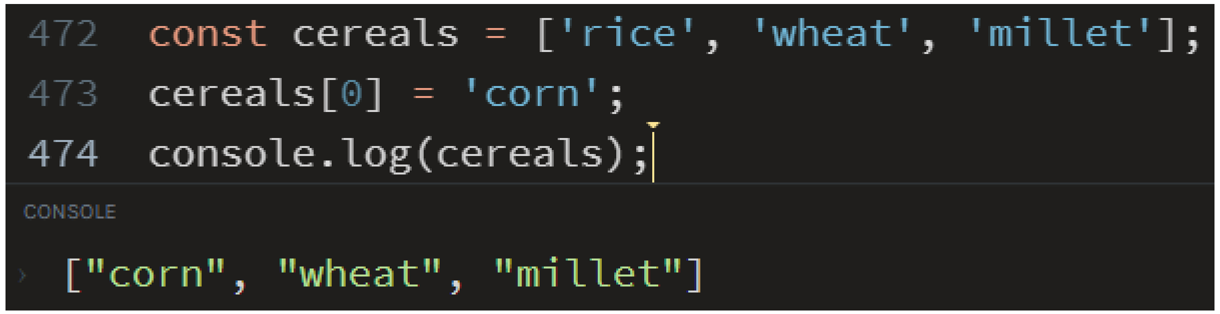
- Now that we have a clearer idea of what is going on in the memory, should we use **const** or **let** when declaring an array?
- First, **const** "blocks" an address in memory, which is why we can't reassign a constant number (or string) to a new value.
- Doing so would imply that we are changing the address in memory.
- But updating an item of an array does not change the address in memory of the array…



```
const pi = 4;
pi = 3.14;
```

CONSOLE

```
TypeError: Assignment
to constant variable.
```

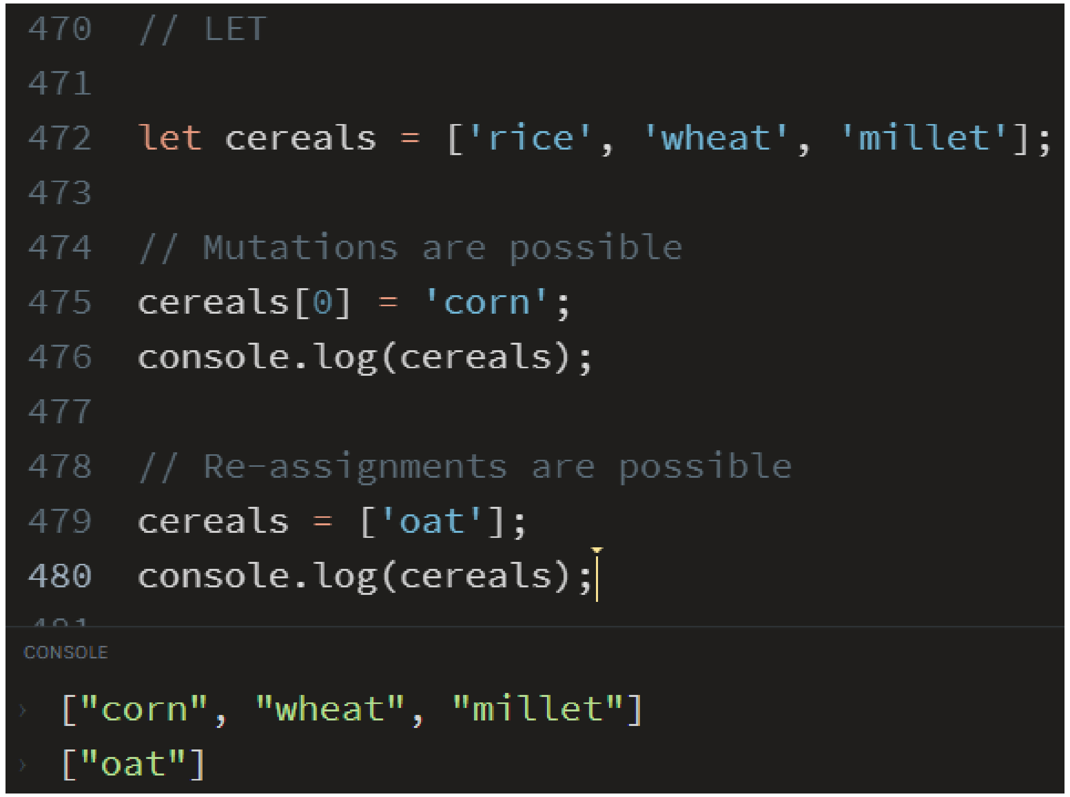


```
const cereals = ['rice', 'wheat', 'millet'];
cereals[0] = 'corn';
console.log(cereals);
```

CONSOLE

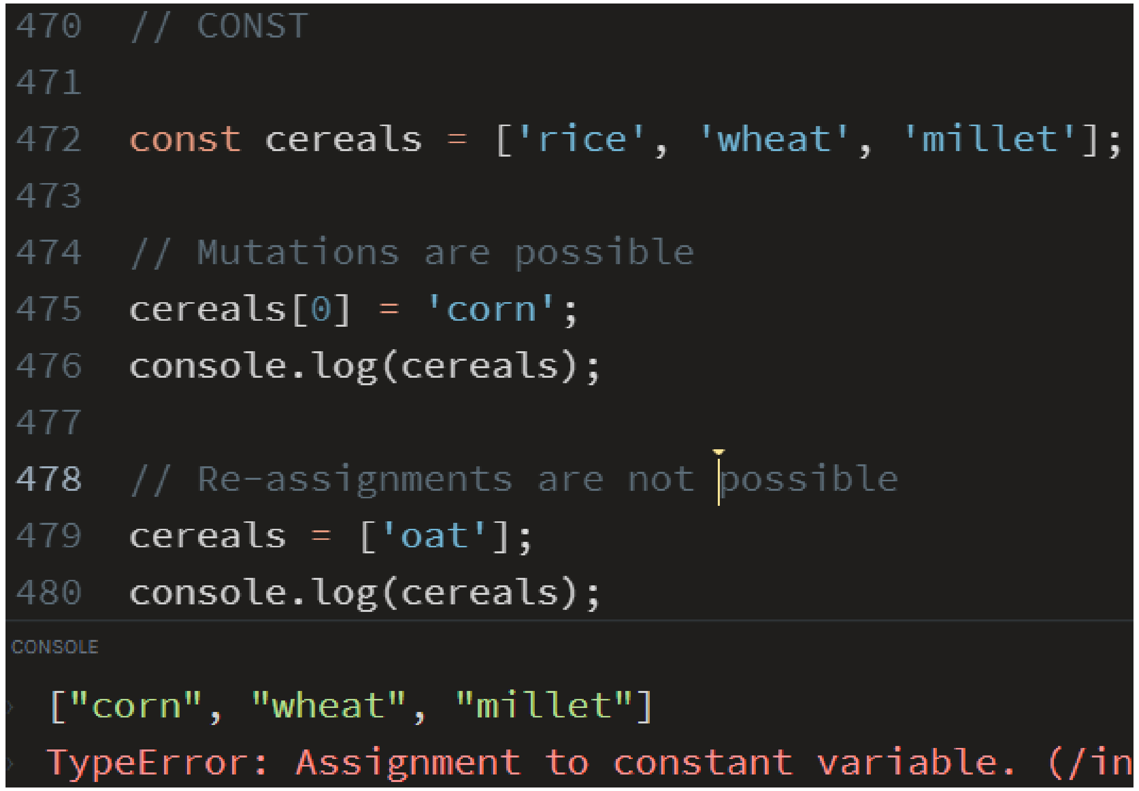
```
["corn", "wheat", "millet"]
```

Let or const for arrays?



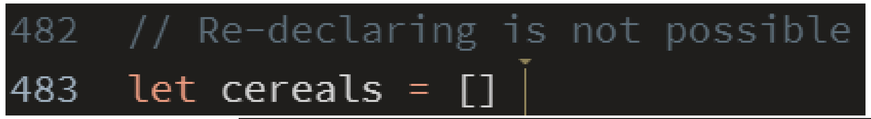
```
// LET

let cereals = ['rice', 'wheat', 'millet'];

// Mutations are possible
cereals[0] = 'corn';
console.log(cereals);

// Re-assignments are possible
cereals = ['oat'];
console.log(cereals);
```

```
CONSOLE
["corn", "wheat", "millet"]
["oat"]
```

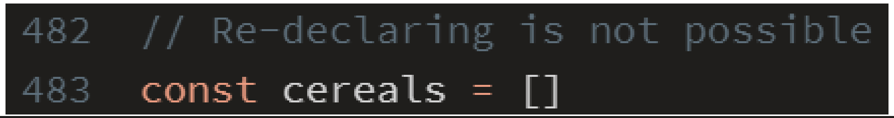


```
// CONST

const cereals = ['rice', 'wheat', 'millet'];

// Mutations are possible
cereals[0] = 'corn';
console.log(cereals);

// Re-assignments are not possible
cereals = ['oat'];
console.log(cereals);
```

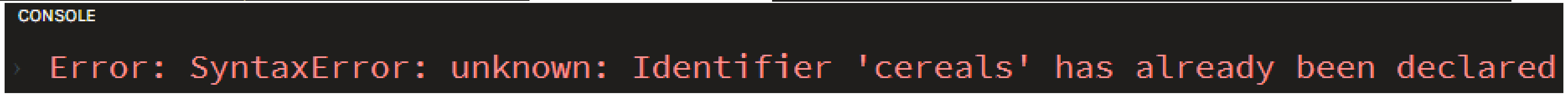
```
CONSOLE
["corn", "wheat", "millet"]
TypeError: Assignment to constant variable. (/in
```



```
// Re-declaring is not possible
let cereals = []
```



```
// Re-declaring is not possible
const cereals = []
```



```
CONSOLE
Error: SyntaxError: unknown: Identifier 'cereals' has already been declared
```

Great advice

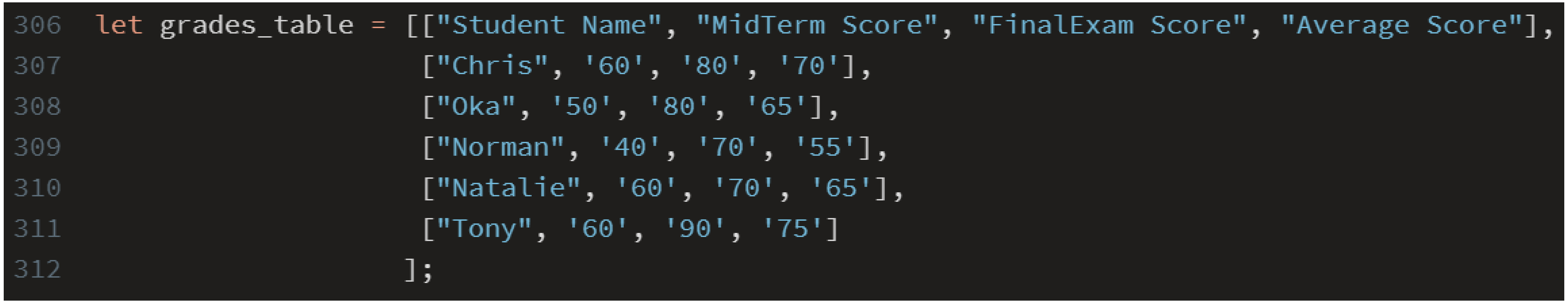
Great Advice #: Prefer const over let when declaring arrays.

It is considered as better practice. Indeed, although mutating or modifying an array (and an object in general) is likely to occur during the execution of a program, re-assigning new values is unlikely and should be avoided anyway.

Activity 6 - Exam adjustments

- Let us assume, that I have **grades** from my students listed in a nested array.

- The first line contains the column labels (student name, some scores) and the other lines will consist of entries regarding some of the students.



```
let grades_table = [["Student Name", "MidTerm Score", "FinalExam Score", "Average Score"],
                     ["Chris", '60', '80', '70'],
                     ["Oka", '50', '80', '65'],
                     ["Norman", '40', '70', '55'],
                     ["Natalie", '60', '70', '65'],
                     ["Tony", '60', '90', '75']
                    ];
```

Activity 6 - Exam adjustments

- Let us assume, that I have **grades** from my students listed in a nested array.
- The first line contains the column labels (student name, some scores) and the other lines will consist of entries regarding some of the students.
- Let us assume that, as a professor, I have decided to be lenient towards my students.
- I realized that the midterm was a bit too difficult compared to last year.
- To compensate for that, I would like to increase the scores of all students on the midterm by 50%.

Activity 6 - Exam adjustments

Write a function
grade_adjustment(),

- which **receives** a grades table, **grades_table**,
- **increases** the **scores** of all students on the **midterm** by **50%**,
- **re-calculates** the **average score**, with the **new adjusted midterm score**,
- and then returns the **updated grades table** as its sole output.

Activity 6 - Exam adjustments

Write a function **grade_adjustment()**,

- which **receives** a grades table, **grades_table**,
- **increases** the **scores** of all students on the **midterm** by **50%**,
- **re-calculates** the **average score**, with the **new adjusted midterm score**,
- and then returns the **updated grades table** as its sole output.

- **Important note:** The **maximal score** for the midterm exam is **capped** to **100**. This means that a student which scores 80 points on the midterm, will not obtain 120 points after the adjustment, but only 100.

Conclusion

- In this lecture, we have covered Arrays in JS, notably:
  - How to create them
  - Most of methods and properties related to arrays
  - The copy problem (alias vs shallow vs deep copies)