[Lesson 9]

Roi Yehoshua 2018

[DAN.IT EDUCATION]

What we learnt last time?

- Working with strings
- Date and Time

[Our targets for today]

- JavaScript Arrays
- Array methods
- Rest and Spread operators
- Iterables
- Set
- Map

[DAN.IT EDUCATION]

Arrays

→ Arrays are used to store lists of related information, e.g., the names of the students in a class, a shopping list, or the grades of your exams

→ The values inside an array are called **elements**

→ There are two syntaxes for creating an empty array:

```
let arr = new Array();
let arr = []; // more common
```

→ We can supply initial elements in the brackets:

```
let fruits = ['Apple', 'Orange', 'Melon'];
```

→ An array can store elements of any type

```
// mix of values
let arr = ['Apple', { name: 'John' }, true, function () { alert('hello'); }];
```

[DAN.IT] EDUCATION

Accessing Array Elements

→ Array elements are numbered, starting with zero

→ We can get an element by its index number in square brackets:

```
let fruits = ['Apple', 'Orange', 'Melon'];

alert(fruits[0]); // Apple
alert(fruits[1]); // Orange
alert(fruits[2]); // Melon
```

→ We can replace an element:

```
fruits[2] = 'Pear'; // now ['Apple', 'Orange', 'Pear']
```

→ Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ['Apple', 'Orange', 'Pear', 'Lemon']
```

→ You can use alert to show the whole array:

```
alert(fruits);
```

[DAN.IT] EDUCATION

Array Length

→ The **length** property of an array returns the the number of array elements:

```
let fruits = ['Apple', 'Orange', 'Melon'];
alert(fruits.length); // 3
```

→ The length property automatically updates when we modify the array

→ The length is actually not the count of values stored in the array, but the greatest numeric index plus one:

```
let fruits = [];
fruits[123] = 'Apple‘;
alert(fruits.length); // 124
```

→ The length property is writable

→ If we increase it manually, nothing happens. But if we decrease it, the array is truncated.

```
let arr = [1, 2, 3, 4, 5];
arr.length = 2; // truncate to 2 elements
alert(arr); // [1, 2]
```

[DAN.IT] EDUCATION

Iterating an Array

→ You can cycle through the array items using a for loop over the indexes:

```
let fruits =  ['Apple', 'Orange',  'Melon'];

for (let i = 0; i < arr.length; i++) {
    alert(arr[i]); // Apple,  Orange, Melon
}
```

→ But for arrays there is another form of loop, **for..of**:

```
for (let fruit of fruits) {
      alert(fruit); // Apple, Orange, Melon
}
```

→ The for..of doesn't give access to the number of the current element, just its value, but in most cases that's enough

Array as a Queue

→ A **queue** is an ordered collection of elements which supports two operations:

→ push appends an element to the end

→ shift gets an element from the beginning, advancing the queue

→ Arrays support both operations:

```
let fruits = ['Apple', 'Orange'];
fruits.push('Melon');
alert(fruits); // Apple, Orange, Melon

alert(fruits.shift()); // remove Apple and alert it
alert(fruits); // Orange, Melon
```

→ The first element added to the queue will be the first one to be removed

→ This makes the queue a **FIFO** (First-In-First-Out) data structure

[DAN.IT] EDUCATION

Array as a Stack

→ There's another use case for arrays – the data structure named **stack**

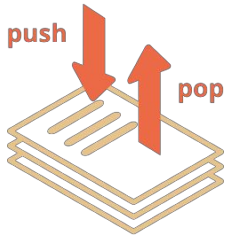
→ It supports two operations:

  → push adds an element to the end

  → pop takes an element from the end

```
let fruits = ['Apple', 'Orange'];
fruits.push('Pear');
alert(fruits); // Apple, Orange, Pear

alert(fruits.pop()); // remove "Pear" and alert it
alert(fruits); // Apple, Orange
```



→ A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top

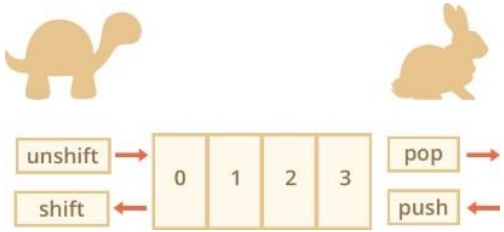
→ For stacks, the latest pushed item is received first

  → This makes the stack **LIFO** (Last-In-First-Out) data structure

[DAN.IT] EDUCATION

[Performance]

→ Methods push/pop run fast, while shift/unshift are slow

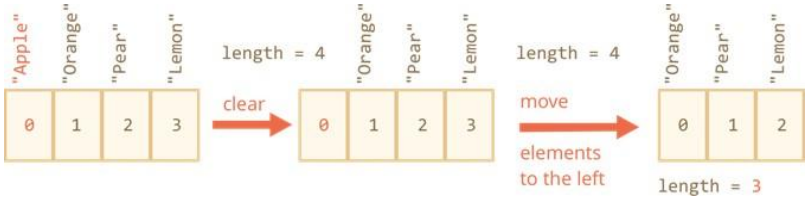


→ For example, the shift operation must do 3 things:

→ Remove the element with the index 0

→ Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on

→ Update the length property

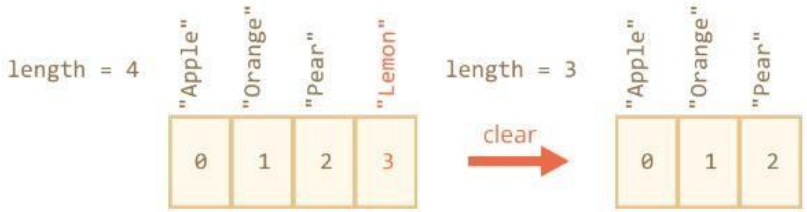


[DAN.IT] EDUCATION

[Performance]

→ On the other hand, push/pop do not need to move anything, because other elements keep their indexes

→ To extract an element from the end, the pop() method cleans the index and shortens length:



Multi-Dimensional Arrays

→ Arrays can have items that are also arrays

→ We can use it for multidimensional arrays, to store matrices:

```
let matrix  = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
alert(matrix[1][1]); // 5
```

[Exercise (1)]

→ Let's try 5 array operations:

- → Create an array styles with items "Jazz" and "Blues"
- → Append "Rock-n-Roll" to the end
- → Replace the value in the middle by "Classics"
  - → Your code for finding the middle value should work for any arrays with odd length
- → Strip off the first value of the array and show it
- → Prepend Rap and Reggae to the array

→ The array in the process:

```
Jazz, Blues
Jazz, Bues, Rock-n-Roll
Jazz, Classics, Rock-n-Roll
Classics, Rock-n-Roll
Rap, Reggae, Classics, Rock-n-Roll
```

[DAN.IT EDUCATION]

Exercise (2)

→ Write a function sumInput() that:

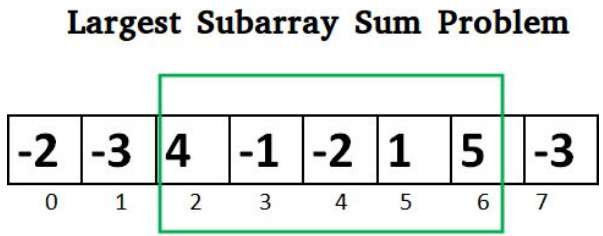
- → Asks the user for values using prompt and stores the values in the array
- → Finishes asking when the user enters a non-numeric value, an empty string, or presses “Cancel”
- → Calculates and returns the sum of array items

[Exercise (3)]

→ The input is an array of numbers, e.g. arr = [-2, -3, 4, -1, -2, 1, 5, -3]

→ Your task is to find the contiguous subarray of arr with the maximal sum of numbers

→ Write the function getMaxSubSu(arr) that will find and return that sum



[DAN.IT EDUCATION]

Rest Parameters …

- → Many JavaScript built-in functions support an arbitrary number of arguments
- → For instance:
  - → Math.max(arg1, arg2, ..., argN) – returns the greatest of the arguments
  - → Object.assign(dest, src1, ..., srcN) – copies properties from src1..N into dest
- → We can define such functions using three dots …
  - → They literally mean “gather the remaining parameters into an array”

```
function sumAll(...args) { // args is the name for the array
    let sum = 0;

    for (let arg of args)
        sum += arg;
    return sum;
}

alert(sumAll(1)); // 1
alert(sumAll(1, 2)); // 3
alert(sumAll(1, 2, 3)); // 6
```

[DAN.IT] EDUCATION

Rest Parameters …

→ We can choose to get the first parameters as variables, and gather only the rest.

→ Here the first two arguments go into variables and the rest go into titles array:

```
function showName(firstName, lastName, ...titles) {
    alert(firstName + ' ' + lastName); // Julius Caesar

    // the rest go into titles array
    // i.e. titles = ["Consul", "Imperator"]
    alert(titles[0]); // Consul
    alert(titles[1]); // Imperator
    alert(titles.length); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

Spread Operator

→ We've just seen how to get an array from the list of parameters

→ But sometimes we need to do exactly the reverse

→ For instance, the function **Math.max()** returns the greatest number from a list:

```
alert(Math.max(3, 5, 1)); // 5
```

→ Now let's say we have an array [3, 5, 1]. How do we call Math.max with it?

→ Passing it "as is" won't work, because Math.max expects a list of numeric arguments

→ The *Spread operator* ...arr "expands" an iterable object arr into the list of arguments

```
let arr = [3, 5, 1];
alert(Math.max(...arr)); // 5 (spread turns array into a list of arguments)
```

[DAN.IT] EDUCATION

Spread Operator

→ We can combine the spread operator with normal values:

```
let arr1  =  [1, -2, 3, 4];
let arr2  =  [8, 3,  -8, 1];

alert(Math.max(1, ...arr1, 2, ...arr2, 25)); // 25
```

→ Also, the spread operator can be used to merge arrays:

```
let merged = [0, ...arr1, 2, ...arr2];
alert(merged); // 0,1,-2,3,4,2,8,3,-8,1 (0, then arr, then 2, then arr2)
```

→ We can use the spread operator with any iterable, not only arrays

→ For instance, we can use it to turn a string into array of characters:

```
let str = "Hello";
alert([...str]); // H,e,l,l,o
```

[DAN.IT] EDUCATION

Additional Array Methods

Method	Description
splice(pos, deleteCount, ...items)	at index pos delete deleteCount elements and insert items
slice(start, end)	creates a new array, copies elements from position start till end (not inclusive) into it
concat(...items)	returns a new array: copies all members of the current one and adds items to it
indexOf/lastIndexOf(item, pos)	look for item starting from position pos, return the index or -1 if not found
includes(value)	returns true if the array has value, otherwise false
find/filter(func)	filter elements through the function, return first/all values that make it return true
sort(func)	sorts the array in-place, then returns it
reverse()	reverses the array in-place, then returns it
split/join	convert a string to array and back
map(func)	creates a new array from results of calling func for every element

Removing Elements from Array

→ The **arr.splice(str)** method is a swiss army knife for arrays

→ It can do everything: add, remove and insert elements

→ The syntax is:

```
arr.splice(index[, deleteCount, elem1, ..., elemN])
```

→ It starts from the position index: removes deleteCount elements and then inserts elem1, ..., elemN at their place. Returns the array of removed elements.

→ Typically it is used for deletion only:

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1); // from index 1 remove 1 element

alert(arr); // ["I", "JavaScript"]
```

[DAN.IT] EDUCATION

Removing Elements from Array

→ The method **arr.slice** is much simpler than similar-looking arr.splice

→ The syntax is:

```
arr.slice(start, end)
```

→ It returns a new array where it copies all items start index "start" to "end" (not including "end")

→ Both start and end can be negative, in that case position from array end is assumed

→ It works like str.slice, but makes subarrays instead of substrings

```
let arr = ["This", "is", "a", "test"];
alert(arr.slice(1, 3)); // is,a
alert(arr.slice(-2)); // a,test
```

[DAN.IT] EDUCATION

Sorting an Array

→ The method arr.sort sorts the array *in place*

```
let arr = [1, 2, 15];
arr.sort();

alert(arr);// 1, 15, 2
```

→ The order became 1, 15, 2. Incorrect. But why?

→ **The items are sorted as strings by default**

→ Literally, all elements are converted to strings and then compared

→ So, the lexicographic ordering is applied and indeed "2" > "15“

→ This is because an array may contain numbers or strings or any type of elements

→ To sort it, we need an *ordering function* that knows how to compare its elements

→ The default is a string order

[DAN.IT EDUCATION]

Sorting an Array

→ To use our own sorting order, we need to supply a function of two arguments as the argument of arr.sort()

→ The function should work like this:

```
function compare(a, b) {
    if (a > b) return 1;
    if (a == b) return 0;
    if (a < b) return -1;
}
```

→ For instance:

```
function compareNumeric(a, b) {
    if (a > b) return 1;
    if (a == b) return 0;
    if (a  < b)  return -1;
}
arr.sort(compareNumeric);

alert(arr);// 1, 2, 15
```

[DAN.IT] EDUCATION

Sorting an Array

→ Actually, a comparison function is only required to return a positive number to say "greater" and a negative number to say "less"

→ That allows to write shorter functions:

```
arr.sort(function (a, b) { return a - b; });

alert(arr);// 1, 2, 15
```

→ Or even shorter using arrow functions:

```
arr.sort((a, b) => a - b);

alert(arr);// 1, 2, 15
```

Searching in Array

→ The methods arr.indexOf(), arr.lastIndexOf() and arr.includes() have the same syntax and do essentially the same as their string counterparts, but operate on items instead of characters

```
let arr = [1, 0, false];

alert(arr.indexOf(0)); // 1
alert(arr.indexOf(false)); // 2
alert(arr.indexOf(null)); // -1

alert(arr.includes(1));  // true
```

→ Note that the methods use === comparison. So, if we look for false, it finds exactly false and not the zero

Searching in Array

→ Say we have an array of objects. How do we find an object with a specific condition?

→ Here the **arr.find()** method comes in handy

→ The syntax is:

```
let result = arr.find(function (item, index, array) {
    // should return true if the item is what we are looking for
});
```

→ For example, we have an array of users, each with the fields id and name

→ Let's find the one with id == 1:

```
let users = [
    { id: 1,  name:  "John" },
    { id: 2,  name:  "Pete" },
    { id: 3, name: "Mary" }
];

let user = users.find(item => item.id == 1);
alert(user.name); // John
```

[DAN.IT] EDUCATION

Searching in Array

→ The find method looks for a single (first) element that makes the function return true

→ If there may be many, we can use **arr.filter(fn)**

→ The syntax is roughly the same as find, but it returns an array of matching elements:

```
let users = [
    { id: 1,  name:  "John" },
    { id: 2,  name:  "Pete" },
    { id: 3, name: "Mary" }
];

// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

[DAN.IT] EDUCATION

Transforming an Array

→ The arr.map method is a useful method for transforming an array

→ The syntax is:

```
let result = arr.map(function (item, index, array) {
    // returns the new value instead of item
})
```

→ It calls the function for each element of the array and returns the array of results

→ For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

Split and Join

→ **str.split(delim)** splits the string into an array by the given delimiter delim

→ In the example below, we split by a comma followed by space:

```
let names = 'Bilbo, Gandalf, Nazgul';
let arr = names.split(', ');

for (let name of arr) {
    alert(`A message to ${name}.`); // A message to Bilbo (and other names)
}
```

→ The call **arr.join(str)** does the reverse to split

→ It creates a string of arr items glued by str between them.

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';');

alert(str); // Bilbo;Gandalf;Nazgul
```

[DAN.IT] EDUCATION

Exercise (4)

→ Write the function sortByName(users) that gets an array of objects with property name and sorts it

→ For instance:

```
let john = { name: "John", age: 25 };
let adam = { name: "Adam", age: 30 };
let mary = { name: "Mary", age: 28 };
let arr = [john, adam, mary];

sortByName(arr);


// now: [adam, john, mary]
alert(arr[1].name); // John
```

Exercise (5)

→ Let arr be an array

→ Create a function unique(arr) that should return an array with unique items of arr

→ For instance:

```
function unique(arr) {
      /* your code */
}
let values = ["John", "Harry", "Mary", "Harry", "Beth", "Harry", "Mary", "John"];

alert(unique(values)); // John, Harry, Mary, Beth
```

[Exercise (6)]

→ You have an array of user objects, each one has name, surname and id

→ Write the code to create another array from it, of objects with id and fullName, where fullName is generated from name and surname

→ For instance:

```
let john = { name: "John", surname: "Smith", id: 1 };
let pete = { name: "Pete", surname: "Hunt", id: 2 };
let mary = { name: "Mary", surname: "Key", id: 3 };

let users = [john, pete, mary];

let usersMapped = /* ... your code ... */

alert(usersMapped[0].id) // 1
alert(usersMapped[0].fullName) // John Smith
```

[DAN.IT] EDUCATION

Iterables

→ **Iterables** are objects that can be used in for..of loops (you can “iterate” over them)

→ Arrays, strings, and many other built-in Javascript objects are iterables

→ Iterables are widely used by the core JavaScript, and many built-in operators and methods rely on them

→ Iterables must implement the method named Symbol.iterator (a special built-in

→ symbol just for that)

→ The result of obj[Symbol.iterator] is an **iterator**, which handles the iteration process

→ An iterator is an object that implements the method next(), which returns an

object {done: Boolean, value: any}

- → **done:true** denotes the iteration end
- → **value** is the next value in the sequence

[DAN.IT] EDUCATION

Iterable Example

→ Let's say we have an object, that is not an array, but looks suitable for for..of

→ Like a range object that represents an interval of numbers:

```
let range = {
    from: 1,
    to: 5
};
// We want the for..of to work:
// for(let num of range) ... num=1,2,3,4,5
```

→ To make the range iterable, we need to add to it a method named Symbol.iterator

→ When for..of starts, it calls that method (or errors if not found)

→ The method must return an *iterator* – an object with the method next()

→ When for..of wants the next value, it calls next() on that object

→ The result of next() must have the form {done: Boolean, value: any}, where done=true means that the iteration is finished, otherwise value must be the new value.

[DAN.IT] EDUCATION

Iterable Example

```
// 1. call to for..of initially calls this
range[Symbol.iterator] = function () {
    // 2. ...it returns the iterator:
    return {
        current: this.from,
        last: this.to,

        // 3. next() is called on each iteration by the for..of loop
        next() {
            // 4. it should return the value as an object {done:.., value :...}  if
            (this.current <= this.last) {
                return { done: false, value: this.current++ };
            } else {
                return { done: true };
            }
        }
    }
};

// now it works!
for (let num of range) {
    alert(num); // 1, then 2, 3, 4, 5
}
```

Calling an Iterator Explicitly

→ Normally, internals of iterables are hidden from the external code

→ There's a for..of loop, that works, that's all it needs to know.

→ But to understand things better, let's see how to create an iterator explicitly

→ We'll iterate over a string the same way as for..of, but with direct calls

```
let str = "hello";

// does the same as
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();
while (true) {
    let result = iterator.next();
    if (result.done) break;
    alert(result.value); // outputs characters one by one
}
```

→ That is rarely needed, but gives us more control over the process than for..of. For example, we can split the iteration process: iterate a bit, then stop, do something else, and then resume later.

[Array.from]

→ The method **Array.from()** takes an iterable and makes a “real” Array from it

→ Then we can call array methods on it, such as push(), pop(), etc.

```
// assuming that range is taken from the example above
let arr = Array.from(range);
arr.push(6);
alert(arr); // 1,2,3,4,5,6
```

→ Here we use Array.from to turn a string into an array of characters:

```
let mystr = '𝒳😍 ';

// splits mystr into array of characters, taking into account surrogate pairs
let chars = Array.from(mystr);

alert(chars[0]); // 𝒳
alert(chars[1]); // 😍
alert(chars.length); // 2
```

→ Unlike str.split, it relies on the iterable nature of string and so, just like for..of, correctly works with surrogate pairs

[DAN.IT] EDUCATION

Set

- Set is a collection of values, where each value may occur only once
- Its main methods are:
  - **new Set**(iterable) – creates the set, optionally from an array of values (any iterable will do)
  - **set.add**(value) – adds a value, returns the set itself
  - **set.delete**(value) – removes the value
    - returns true if value existed at the moment of the call, otherwise false
  - **set.has**(value) – returns true if the value exists in the set, otherwise false
  - **set.clear**() – removes everything from the set
  - **set.size** – the elements count

Set Example

→ For example, we'd like to store all the users who have visited our site

→ But repeated visits should not lead to duplicates (a visitor must be counted only once)

→ Set is just the right thing for that:

```
let set = new Set();
let john = { name: "John" };
let peter = { name: "Peter" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(peter);
set.add(mary);
set.add(john);
set.add(mary);

// set keeps only unique values
alert(set.size); // 3

for (let user of set) {
    alert(user.name); // John (then Peter and Mary)
}
```

Exercise (7)

→ Let arr be an array

→ Create a function unique(arr) that should return an array with unique items of arr

→ Use set to make the function more efficient

→ For instance:

```
function unique(arr) {
      /* your code */
}
let values = ["John", "Harry", "Mary", "Harry", "Beth", "Harry", "Mary", "John"];

alert(unique(values)); // John, Harry, Mary, Beth
```

Exercise (8)

→ Write a function subArrayZero(arr) that gets an array and returns whether it contains a contiguous subarray whose sum is equal to 0

→ Your function should go over the array elements only once

```
function subArrayZero(arr) {
    // your code
}

alert(subArrayZero([-5, 12, 4, -7, 2, 1, 8])); // true, 4 + (-7) + 2 + 1 = 0
alert(subArrayZero([3, -2, -6, 2, 1, -2])); // false
```

[DAN.IT] EDUCATION

Map

- Map is a collection of keyed data items, just like an Object
- The main difference is that Map allows keys of any type
  - Objects can also be keys
- The main methods are:
  - **new Map()** – creates the map.
  - **map.set**(key, value) – stores the value by the key and returns the map
  - **map.get**(key) – returns the value by the key, undefined if key doesn't exist in map
  - **map.has**(key) – returns true if the key exists, false otherwise
  - **map.delete**(key) – removes the value by the key
  - **map.clear**() – clears the map
  - **map.size** – returns the current element count

Map Examples

```
let map = new Map();
map.set('1', 'str1');  // a string key
map.set(1, 'num1');    // a numeric key
map.set(true, 'bool1'); // a boolean key

// Map keeps the key type (unlike Object), so these two are different:
alert(map.get(1));      // 'num1'
alert(map.get('1'));    // 'str1'
alert(map.size);        // 3
```

```
// Using objects as keys
let user = { name: "John" };

// for every user, let's store his visits count
let visitsCountMap = new Map();

// john is the key for the map
visitsCountMap.set(user, 123);

alert(visitsCountMap.get(john)); // 123
```

[DAN.IT] EDUCATION

Map From Object

→ When a Map is created, we can pass an array (or another iterable) with key-value pairs, like this:

```
let map = new Map([
    ['1', 'str1'],
    [1, 'num1'],
    [true, 'bool1']
]);
```

→ There is a built-in method Object.entries(obj) that returns an array of key/value pairs for an object exactly in that format

→ So we can initialize a map from an object like this:

```
let map = new Map(Object.entries({
    name: "John",
    age: 30
}));
```

[DAN.IT EDUCATION]

Iteration over Maps

→ For looping over a map, there are 3 methods:

- → **map.keys()** – returns an iterable for keys
- → **map.values()** – returns an iterable for values
- → **map.entries()** – returns an iterable for entries [key, value]
  - →It is used by default in for..of

```
let recipeMap = new Map([
        ['cucumber', 10],
        ['tomatoes', 15],
        ['onion', 3]
]);

// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
        alert(vegetable); // cucumber, tomatoes, onion
}

// iterate over values (amounts)
for (let amount of recipeMap.values()) {
        alert(amount); // 10, 15, 3
}

// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of
recipeMap.entries()
        alert(entry); // cucumber,10 (and so on)
}
```

[Exercise (9)]

→ Create a function countWords(sentence) that gets a sentence and prints to the console the number of occurrences of each word in the sentence

→ For instance:

```
function countWords(sentence) {
    // your code
}

let sentence = "John the second is the son of John the
first,  while the second son of John the second is William
the  second.";
countWords(sentence);
```

```
John 3
the 6
second 4
is 2
son 2
of 2
first 1
while 1
William 1
```

[DAN.IT] EDUCATION

Exercise (10)

→ Anagrams are words that have the same number of same letters, but in different order

→ For instance:

- → nap - pan
- → ear - are - era
- → cheaters - hectares – teachers

→ Write a function aclean(arr) that returns an array cleaned from anagrams

→ For instance:

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];

alert(aclean(arr)); // "nap,teachers,ear" or "PAN,cheaters,era"
```

- → From every anagram group should remain only one word, no matter which one

[DAN.IT] EDUCATION

[Control questions]

1. What is array?
2. How can we create an array in Javascript?
3. What array methods do you know?
4. What is “rest” operator?
5. What is “spread” operator?
6. How do we sort an array?
7. How can we find an item in the array?
8. What is “iterable” in JavaScript?
9. What is Set?
10. What is the difference between Array.push and Set.add?
11. What is Map?
12. What can be used as Map key?

[DAN.IT] EDUCATION