In this post, we will take a look at different data structures that are available to us out of the box in JavaScript (out of the box since you don't have to code these data structures yourself). We will also take a look at a few methods that will help us process these data structures.

JavaScript has built in support for the following data structures:

1. Array
2. Set
3. Map

**Array**

An array is a collection of items. In JavaScript, you can have an array that contains different types of data. In practice, you will encounter arrays with same data type though.

There are three ways to create an array.

1. Define a variable and assign it
2. new Array([1....n])
3. new Array(array length)

**Array operations**

* Accessing elements from the array:  
  Array elements have an index, index starts from 0 and goes to array.length -1. To access an element at any particular index use the following syntax arrName[index].
* To add elements to an already existing array use array.push().
* To find the number of elements in the array use array.length property.
* To check if an element exists use array.indexOf() or array.includes().
* To remove an element from an array, we can use array.splice() method. If you want to remove the last element you can use array.pop() method.  
  Splice modifies the existing array, to remove an element at particular index use array.splice(index, 1).
* You can combine two arrays using array.concat() method.

**Set**

Like Array, Set is also a collection of items, the difference being in a set a value will appear only once.

You can create a set as follows:

let set = new Set([1, 2, 3, 4, 5, 6, 6, 6])

set will contain 1,2,3,4,5,6

**Set operations**

* To get the number of elements in the set use the size property of the Set (set.size).
* To add value use set.add(), returns the set.
* To remove an element use set.delete(value), to remove all the elements from the set use set.clear()

Set is useful when you need a unique collection of items if you used an array for the same task you will need to have extra logic that eliminates the duplicates.

**Map**

Is a collection of key-value pairs, so is it the same as an object?  
At the top level, they appear to be the same however, there are few differences.

* Keys of an object can only be Strings, that's not the case for maps.
* Keys are stored in insertion order in a Map.
* You can also get the size of the Map with size property.
* A Map is iterable, so you can directly use .forEach on Map. For objects, you will have to get the keys first to get that key's value.
* Along with these, Map prototype has a few helper methods.  
  For example, to check if a key is present in the map you could do  
  map.has(key).

You can create a Map as follows:

let map = new Map([['name', 'John'], ['age', '21']])

**Map operations**

* To get number of elements in the Map use the size property.
* You can't access values from the map as you would do with an Object. You need to use .get() method on the map.

To add value to the map you need to use the .set() method.

Now that you know the basics of these data structures go ahead and try them out, implement something.

**References**

* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/pop>
* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols>
* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set>
* <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map>

**SET**

  //set

      let set = new Set();

      //add

      set.add(1);

      set.add(1);

      set.add(2);

      set.add(3);

      //check availability

      console.log(set.has(1));

      console.log(set.has(4));

      //check size

      console.log(set.size);

      console.log(set);

      set.forEach(element => {

        console.log(element);

      });

      //clear total set

      set.clear();

      console.log(set);

      // it contains

      // ["sumit","amit","anil","anish"]

      var set1 = new Set(["sumit", "sumit", "amit", "anil", "anish"]);

      // it contains 'f', 'o', 'd'

      var set2 = new Set("fooooooood");

      // it contains [10, 20, 30, 40]

      var set3 = new Set([10, 20, 30, 30, 40, 40]);

      // it is an  empty set

      var set4 = new Set();

**Map**

<script>

      let map = new Map();

      map.set("ishan", 23);

      map.set("nishan", 22);

      map.set("milshan", 23);

      map.set("kishan", 24);

      map.set("hishan", 26);

      map.set("ashan", 25);

      if (map.has("ishan")) {

        console.log(map.get("ishan"));

      }

      map.delete("ishan");

      console.log(map.get("ishan"));

      for (const [key, val] of map.entries()) {

        console.log(key, val);

      }

      for (const key of map.keys()) {

        console.log(key);

      }

      for (const val of map.values()) {

        console.log(val);

      }

      map.forEach(element => {

        console.log(element);

      });

    </script>

## Before ES6

[ECMAScript](https://flaviocopes.com/ecmascript/) 6 (also called ES2015) introduced the Map data structure to the [JavaScript](https://flaviocopes.com/javascript/) world, along with [Set](https://flaviocopes.com/javascript-data-structures-set/)

Before its introduction, people generally used objects as maps, by associating some object or value to a specific key value:

const car = {}

car['color'] = 'red'

car.owner = 'Flavio'

console.log(car['color']) //red

console.log(car.color) //red

console.log(car.owner) //Flavio

console.log(car['owner']) //Flavio

## Enter Map

ES6 introduced the Map data structure, providing us a proper tool to handle this kind of data organization.

A Map is initialized by calling:

const m = new Map()

### Add items to a Map

You can add items to the map by using the set method:

m.set('color', 'red')

m.set('age', 2)

### Get an item from a map by key

And you can get items out of a map by using get:

const color = m.get('color')

const age = m.get('age')

### Delete an item from a map by key

Use the delete() method:

m.delete('color')

### Delete all items from a map

Use the clear() method:

m.clear()

### Check if a map contains an item by key

Use the has() method:

const hasColor = m.has('color')

### Find the number of items in a map

Use the size property:

const size = m.size

## Initialize a map with values

You can initialize a map with a set of values:

const m = new Map([['color', 'red'], ['owner', 'Flavio'], ['age', 2]])

## Map keys

Just like any value (object, array, string, number) can be used as the value of the key-value entry of a map item, **any value can be used as the key**, even objects.

If you try to get a non-existing key using get() out of a map, it will return undefined.

## Weird situations you’ll almost never find in real life

const m = new Map()

m.set(NaN, 'test')

m.get(NaN) //test

const m = new Map()

m.set(+0, 'test')

m.get(-0) //test

## Iterating over a map

### Iterate over map keys

Map offers the keys() method we can use to iterate on all the keys:

for (const k of m.keys()) {

console.log(k)

}

### Iterate over map values

The Map object offers the values() method we can use to iterate on all the values:

for (const v of m.values()) {

console.log(v)

}

### Iterate over map key, value pairs

The Map object offers the entries() method we can use to iterate on all the values:

for (const [k, v] of m.entries()) {

console.log(k, v)

}

which can be simplified to

for (const [k, v] of m) {

console.log(k, v)

}

## Convert to array

### Convert the map keys into an array

const a = [...m.keys()]

### Convert the map values into an array

const a = [...m.values()]

## WeakMap

A WeakMap is a special kind of map.

In a map object, items are never garbage collected. A WeakMap instead lets all its items be freely garbage collected. Every key of a WeakMap is an object. When the reference to this object is lost, the value can be garbage collected.

Here are the main differences:

1. you cannot iterate over the keys or values (or key-values) of a WeakMap
2. you cannot clear all items from a WeakMap
3. you cannot check its size

A WeakMap exposes those methods, which are equivalent to the Map ones:

* get(k)
* set(k, v)
* has(k)
* delete(k)

The use cases of a WeakMap are less evident than the ones of a Map, and you might never find the need for them, but essentially it can be used to build a memory-sensitive cache that is not going to interfere with garbage collection, or for careful encapsualtion and information hiding.

[JavaScript](https://hackernoon.com/tagged/javascript) ES6 introduces a new [data structure](https://hackernoon.com/tagged/data-structure), called *maps*. Mapsaredesigned as an alternative to using Object literals for storing key/value pairs that require unique keys, and provide very useful methods for iteration.

**Object literals as “Maps”**

Nothing is more fundamental in JavaScript than object literals. Creating a map of sorts is as simple as declaring it in code.

var map = {  
 "key1":"value1",  
 "key2":"value2"  
}  
map.key1; // == "value1"

This will work in many situations, but there are problems with using object literals as maps. We may have keys that clobber Object prototype methods.

var map = {  
 "key1":"value1",  
 "key2":"value2"  
}  
map.toString() == "[object Object]" // Correct  
map.toString = "value3"  
map.toString(); // Uncaught TypeError: map.toString is not a function

Another drawback of using object literals is all keys can only be strings. This works in many situations, but when attempting to use a primitive value as a string, the system will convert it to a string behind the scenes.

This kind of magic which happens under the hood and without notification to the user can cause unexpected results, for instance, if the provided key is an array.

map[[1,2,3]] = "value4" // the provided key is an array  
map['1,2,3'] = "value4" // the system has made the key a string

Another problem with using object literals (pre-ES6), is property/key orders are not guaranteed. Just because you have added the keys in a certain order, does not mean they will remain in that order, when you iterate through the keys.

var o = {}  
o.a = 1  
o.b = 2  
o.c = 3

for(key in o) console.log(key);   
// expected a,b,c - but not guaranteed to be in that order

Objects also lacks a forEach method. If you are used to iterating arrays using .forEach(), objects cannot be iterated in this way.

o.forEach // undefined

**Maps in ES6**

ES6 Maps provide a new object for this purpose. You create a new map by calling **new Map([iterable])**. Iterable can be an array or any iterable object whose elements are key/value pairs. The provided key/value pairs are added to the new map.

var m = new Map()  
m.set('a', 1)  
m.set('b', 2)  
// {"a" => 1, "b" => 2}

**Map Properties and Methods**

Map provides a very convienent, **.size** property to get the size of the map. The size is also very convienently shown in the Chrome Dev console, along with the contents of the map.

var m = new Map()  
m.set('a', 1)  
m.set('b', 2)  
m.size; // 2 Make sure to use .set so that size updates correctly

m; // Map(2) {"a" => 1, "b" => 2}

**map.clear()**

Clears the map of all entries.

**map.delete(key)**

Deletes a key and returns if the delete was successful.

var m = new Map()  
m.set('a', 1)  
m.set('b', 2)  
m.delete('a'); // true  
m.delete('c'); // false (key was not there to delete)

**map.get(key), map.has(key)**

Getting a key, and finding if a key is present.

m.get('a'); // 1  
m.has('a'); // true  
m.has('z'); // false

**map.forEach(fn)**

Provides a convienient way to iterate the map. You can get keys, values, and the map itself very easily.

var m = new Map()  
m.set('a', 1)  
m.set('b', 2)  
m.forEach((k, v, m) => console.log(`key:${k} value:${v} map:${m}`))

// key:1 value:a map:[object Map]  
// key:2 value:b map:[object Map]

**for..of**

Another option to iterate the map is to use **for..of**syntax, which can easily provide access to the keys and values of the map.

for([key,value] of m)   
 console.log(key + '=' + value)  
// a=1  
// b=2

**m.keys()**

Returns a full-blown iterator, so you can iterate the keys one by one, on demand, using .next()

var m = new Map()  
m.set('a', 1)  
m.set('b', 2)

var iter = m.keys()  
iter.next(); // Object {value: "a", done: false}  
iter.next(); // Object {value: "b", done: false}  
iter.next(); // Object {value: *undefined*, done: true}

**m.values(), m.entries()**

The map Values and Entries can be iterated in the same way. Entries will give you an array of the kind [key, value].

var iter = m.values()  
iter.next(); // Object {value: 1, done: false}  
iter.next(); // Object {value: 2, done: false}  
iter.next(); // Object {value: *undefined*, done: true}

var iter = m.entries()  
iter.next(); // Object {value: ["a", 1], done: false}  
iter.next(); // Object {value: ["b", 2], done: false}  
iter.next(); // Object {value: *undefined*, done: true}

Maps give you alot of control over operations that need to be performed on the keys, values, or entries.

**2d Arrays to Maps**

Maps can take 2d arrays in the constructor. Each array entry should have the format [key, value].

var arr = [['a', 1], ['b', 2]]  
var m = new Map(arr)

m; // Map(2) {"a" => 1, "b" => 2}

**Non-Strings as keys**

You can use objects of all kinds as map keys. The system will not automatically convert the keys to strings as it does for object literals. This opens up a wide range of opportunities to do interesting things with maps. For instance, setting the document object as a key.

var m = new Map()

m.set(document, true) // uses the document object as a key  
// Map(1) {#document {} => true}

m.get(document)   
// true

Maps are an important new feature of ES6, and can be used in a wide variety of use cases for storing key/value pairs.