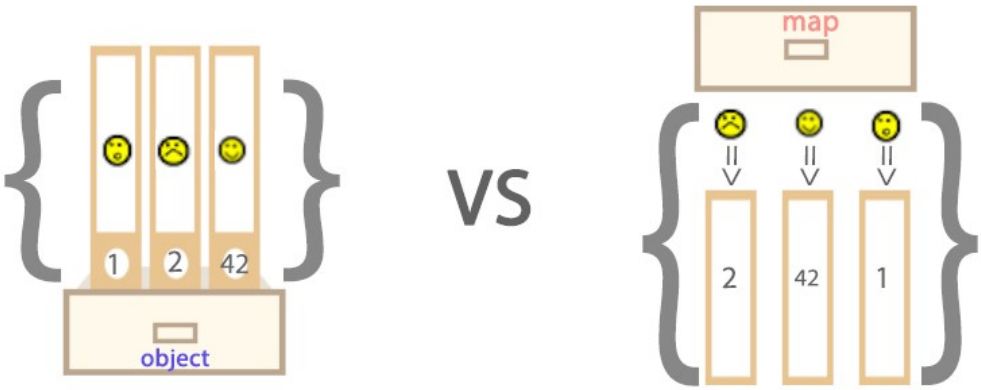
ES6 — Map vs Object — What and when?

Maya Shavin
Feb 1, 2018 · 10 min read



Object vs Map in a happy way

You may wonder — why Map vs Object but not Map vs Array, or Object vs Set? Well, you can also compare between any of the two, but Map and Object, unlike others, have very similar use-cases which require us to understand more deeply about them to decide what is better for when. And that's what this is about.

Let's start, shall we?

. . .

The concepts

What is Map?

Map sounds very simple, doesn't it? We see or hear about it almost everyday, let's say World map, Street map, etc…. So what is exactly Map?

Map is a data collection type (in a more fancy way — *abstract data structure type*), in which, data is stored in a form of **pairs**, which contains a **unique key** and **value mapped to that key**. And because of the uniqueness of each stored key, there is **no duplicate** pair stored.

You may recognize by now one common thing about all examples mentioned above — they are used for looking up something (can be a country — for World map, a street name — Street map, etc…).

That's right, Map is mainly used for **fast searching and looking up** data.

Example: {(1, "smile"), (2, "cry"), (42, "happy")}

in which each pair is in the format — (key, value).

Important note: **key and value in Map can be in any data type,** *not limited* to only string or integer.

And what about Object?

Everyone knows Object, especially in Javascript. Object is object, isn't it? Right, but not enough.

***Regular* Object** (pay attention to the word '*regular*') in Javascript is *dictionary type* of data collection — which means it also **follows key-value stored concept** like Map. Each **key** in Object — or we normally call it "property" — is also **unique** and **associated with a single value**.

In addition, **Object** in Javascript has **built-in prototype**. And don't forget, *nearly all objects in Javascript are **instances of Object**, including Map.*

Example: {1: 'smile', 2: 'cry', 42: 'happy'}

Therefore, by definition, **Object** and **Map** are based on the *same* concept — using key-value for storing data. However, like we always say — *same same but different* — they are indeed quite different from each other, mainly in:

- *Key field*: in **Object**, it *follows the rule of normal dictionary*. The keys **MUST be simple types** — either *integer* or *string or symbols*. Nothing more. But in **Map** it can be **any data type** (an object, an array, etc…). (*try using another object as Object's property key — i dare you :)*)
- *Element order:* in **Map**, original **order of elements (pairs) is preserved**, while in **Object**, it **isn't**.
- *Inheritance*: **Map** is an instance of Object (*surprise surprise!*), but **Object** is definitely **not** an instance of Map.

```
var map = new Map([[1,2],[3,4]]);
console.log(map instanceof Object); //true

var obj = new Object();
console.log(obj instanceof Map); //false
```

But not only those. What else makes them different from each other? Let us continue.

How to construct

Object

Like **Array**, **Object** is straight-forward. To declare **new object**, all you need to do is using direct literal:

```
var obj = {}; //Empty object
var obj = {id: 1, name: "Test object"};
//2 keys here: id maps to 1, and name maps to "Test object"
```

Or by constructor

```
var obj = new Object(); //Empty Object
var obj = new Object; //same result
```

Or by using *Object.prototype.create*

```
var obj = Object.create(null); //Empty Object
```

***Special note*:**

You only should use **Object.create** in very *specific* cases, such as:

- You want to choose prototype object to inherit to, without the need to define constructor. It's a way of "inheritance".

```
var Vehicle = {
    type: "General",
    display: function(){console.log(this.type);}
}

var Car = Object.create(Vehicle); //create a new Car inherits from
Vehicle
Car.type = "Car"; //overwrite the property
Car.display();//"Car"

Vehicle.display();//still "General"
```

In general, like in **Array**, **do not** use built-in constructor over literal in creating new object, because:

- *More* typing
- *Slower* performance (much much slower)
- Confusion & increasing *more chances for mistake*, for example:

```
var obj = new Object(id: 1, name: "test") //Error - obviously

var obj1 = {id: 1, name: "test"};
var obj2 = new Object(obj1); //obj1 and obj2 points to the same one
obj2.id = 2;
console.log(obj1.id); //2
```

And in any case, who wants to type extra unnecessary code — at all?

Map

Map, in the other hand, has only one way to create, by using its built-in constructor and ***new*** syntax.

```
var map = new Map(); //Empty Map
var map = new Map([[1,2],[2,3]]); // map = {1=>2, 2=>3}
```

> *Map([iterable])*

The constructor receives an **array** or **iterable** object whose elements are key-value pairs — aka ***arrays*** *with 2 elements [key, value]*.

So far so good? Great. Now it's time to move on to our next step — comparison between **Map/Object's** basic capabilities, which are:

Accessing element

- For **Map**, accessing an element value is done by *Map.prototype.get(key)* — which means we *need to know the key* to be able to retrieve the value of an element

```
map.get(1)// 2
```

- Similarly in **Object**, we do need to know the key/property in order to get the element value, but in different syntax: *Object.<key>* and *Object['key']*:

```
obj.id //1
obj['id'] //1
```

- Checking if a key is already in **Map** is supported by

```
map.has(1);//return boolean value:  true/false
```

- While in **Object**, we need to do a bit extra

```
var isExist = obj.id === undefined; //check if obj has that property
defined before.
```

- Or

```
isExist = 'id' in obj; //which will apply for inherited property as
well.
```

The syntax in **Map** is simpler and more direct than in **Object** in this case.

Note: in **Object**, we have *Object.prototype.hasOwnProperty()* will return *true/false* to check if it has a specified key as its **own** property — it will be good in case **we only check a key which is not inherited for that object**. Still, to my opinion, here **Map** wins over **Object** regarding the ease of usage.

Adding new element

- **Map** supports adding new element by providing *Map.prototype.set()* which takes 2 parameters: key, value.

```
map.set(4,5); //{1=>2, 2=>3, 4=>5}
```

- But if you pass an *existing key*, it will ***overwrite*** the value mapped to that key with the new value — as what set operation is supposed to do.

```
map.set(4,6); //{1=>2, 2=>3, 4=>6}
```

- Similarly, adding new set of property to **Object** is done directly by:

```
obj['gender'] = 'female'; //{id: 1, name: "test", gender: "female"}
obj.gender = male;
//Both is ok and will overwrite the existing mapped value if
property already exists.
//{id: 1, name: "test", gender: "male"}
```

- As you can see, both performs theoretically in ***O(1)*** running time for adding thanks to its structure, so retrieving a key doesn't require to scan through all of data. What about removing/deleting?

Removing/Deleting an element

In **Object**, there is no built-in method to delete a property from it. Instead, we can use the operator **delete** as:

```
delete obj.id; //{name: "test", gender: "male"}
```

Pay attention that some may argue about whether it's better to do the following instead, for performance boost.

```
obj.id = undefined;
```

However, the logic is quite different here:

- ***delete*(key)** will **remove that specific property completely** from object, but

- setting "*obj[key] = undefined*" actually **just changed the mapped value of that property to "undefined"**, and that property still remain on its place in that object.

Therefore, when we use *"for…in…"*, we *still iterate over that property's key* regardless.

And of course, *the check for whether a key/property already exists* in an obj will yield **two different results** in these two scenarios, except the following check:

```
obj.id === undefined;//same result
```

Hence, think carefully. Performance boost sometimes is not worth it! :)

Oh one more thing, **delete** operator does return a string of "*true"/ "false",* but unlike normal, this returned value indicates quite differently status, in which:

- *true* for all cases **except** when the property is an **own non-configurable property.**
- Otherwise, *false* for non-strict mode, and *Exception error* will be thrown in strict mode instead.

Meanwhile, Map, again, has few nice built-in methods to support different removing purposes, such as:

- ***delete(key)*** for removing a target element with a *specified key* from a map. Don't forget, this **delete()** returns a **boolean** variable, indicate if target element with specific key *did exist* in the map and *was removed* successfully (*true*) or if that target element *does not exist* at all in map (*false*).

```
var isDeleteSucceeded = map.delete(1); //{ 2=>3, 4=>5}
console.log(isDeleteSucceeded); //true
```

- ***clear()*** — remove **ALL elements** from a Map object.

```
map.clear(); //{}
```

- In order to achieve the same capability of ***clear***(), Object will need to iterate through its properties (keys) and delete one by one until the end. This can be tiring, especially when we feel a little bit of lazy (just a tiny bit :))

In general, both Map and Object performance in removing element is pretty similar to each other, again due to its structure. **Deleting** a key will take **O(1)**, while **clearing out all elements will still take O(*n*)** with *n* is the size of Map/Object. *So, yeah, it's definitely a tie here!*

Since we mentioned a bit about the size, let's see how **Map**/**Object** behaves:

Getting the size

One advantage here, which **Map** has in comparison to **Object**, is that **Map** does keep its size updated automatically, and we can always get the size easily by:

```
console.log(map.size);//0
```

While with **Object**, it needs to be calculated manually, with the help of *Object.keys()* — which returns an array of all existing keys in a certain object:

```
console.log(Object.keys(obj).length);//2
```

Got a hang of it? Great. Now the last comparison, as it marks one of the significant differences in Map and Object — iteration between elements.

Iterating

Map is built-in iterable — **Object** is not. Simple as that.

- ***Bonus****: how do you check if a type is iterable? By using*

```
//typeof <obj>[Symbol.iterator] === “function”
console.log(typeof obj[Symbol.iterator]); //undefined
console.log(typeof map[Symbol.iterator]); //function
```

Which means, in **Map** all elements can be iterated directly with "*for… of*" as:

```
//For map: { 2=>3, 4=>5}
for (const item of map){
    console.log(item);
    //Array[2,3]
    //Array[4,5]
}

//Or
for (const [key,value] of map){
    console.log(`key: ${key}, value: ${value}`);
    //key: 2, value: 3
    //key: 4, value: 5
}
```

Or with its built-in **forEach()**:

```
map.forEach((value, key) => console.log(`key: ${key}, value:
${value}`));
//key: 2, value: 3
//key: 4, value: 5
```

But with **Object**, either we use "*for… in*"

```
{id: 1, name: "test"}
for (var key in obj){
   console.log(`key: ${key}, value: ${obj[key]}`);
   //key: id, value: 1
   //key: name, value: test
}
```

Or using **Object.keys(obj)** to get all the keys and iterate:

```
Object.keys(obj).forEach((key)=> console.log(`key: ${key}, value:
${obj[key]}`));
//key: id, value: 1
//key: name, value: test
```

OK, here comes the question — since they are really similar to each other in both structure and performance wise, with Map having ***slightly more*** *advantages compared to Object, should we always prefer Map over Object?*

When to use Map? And when to use Object?

Again, despite all the advantages **Map** can have against **Object**, there is still cases **Object** will perform better. After all, Object is the most basic concept of Javascript.

- **Object** is the great choice for scenarios when we only need simple structure to store data and knew that all the keys are either strings or integers (or Symbol), because creating plain Object and accessing Object's property with a specific key is much **faster** than creating a Map (*literal vs constructor, direct vs get() function call — you know who wins already*).
- Also, in scenarios where there is a need to apply separate logic to individual property/element(s), then **Object** is definitely the choice. For example:

```
var obj = {
    id: 1,
    name: "It's Me!",
    print: function(){
        return `Object Id: ${this.id}, with Name: ${this.name}`;
    }
}

console.log(obj.print());//Object Id: 1, with Name: It's Me.
```

(*Try to do it the same with Map. You just can't!*)

- Moreover, *JSON has direct support* for **Object**, but not with **Map** (yet). So in certain situation where we have to work a lot with JSON, consider Object as preferred option.

- Otherwise, **Map** is purely hash, **Object** is more than that (with support inner logic). And using **delete** operator with Object's property has *several performance issues* (we will discuss about this in a different article). Hence in scenarios that requires a lot of adding and removing (especially) new pair, **Map** may perform much better.
- In addition, **Map** preserves the order of its keys — unlike **Object**, and **Map** was built with iteration in mind, so in case iteration or elements order are highly significant, consider **Map** — *it will ensure stable iteration performance in all browsers*.
- And last but not least, **Map** tends to perform better in storing large set of data, especially when *keys are unknown until run time, and when all keys are the same type and all values are the same type*.

. . .

Conclusion

In conclusion, it really depends on what kind of data (input) you are going to work with and what operations you are going to perform on them in order to prefer one to the other between **Map** and **Object**.

Map tends to have more advantages over **Object** in scenarios when we just need a simple look-up structure for data storing, with all the basic operations it provided. However, **Map** can't never replace **Object**, in any sense, because in Javascript, **Object** is — after all — **more than just** a normal hash table (*and therefore shouldn't be used as a normal hash table if there is alternative, it's just a waste of a great resource ;)*).

Now honestly, which one you like better? :). Tell me in comments. I'd love to know your opinion.

. . .

More on ES6:

- Set vs Array — What and when?
- Spread (…) syntax and its magic
- ES6 Cool stuffs — var, let and const in depth

- ES Cool stuffs — Destructuring statement in depth
- Template literals in depth
- Let’s divide our phones into Classes

Coming up next —ES6 Promise (or something else interesting, I promise!)… Until then, see you soon :)!

If you like this post and want to read more, feel free to check out my articles.

If you’d like to catch up with me sometimes, follow me on Twitter | Facebook or simply visit my portfolio website.