

Rocking JS data structures!

areknawo.com/rocking-js-data-structures

JavaScript's development has been quite stubborn up to 2015. Yeah, that's the magic time **ES6** was announced and the whole web-development-thing really took off and grew in popularity exponentially. ☐ But, that's something every JS fan probably knows - the year, the exact moment has been repeatedly referenced in many, many JS resources around the world. So, let's be innovative and do the same again, shall we? 😊

ES6 has brought a great number of **new goodies** to JS. Not only now-must-have **arrow functions**, **promises**, and syntactic sugar, but also new **data structures**. 📦 That's right, I'm talking about things like Sets, WeakMaps and etc. (if you already know them). These little, but very interesting features have been pushed into the background, mainly because of how long it took for modern browsers to fully embrace new specification. As time passed ⌚, people started using new syntax and some really desired new functionalities, but these structures became **less relevant**. Of course not to all, but taking even as obvious example as myself - I hardly ever used them. I just stuck with old-school arrays and object and lived within that limited scope. But, don't worry, because in this article we'll explore how good and useful these structures can really be. With new **possibilities** they provide and their current support... just why not? 😊

TypedArrays

I guess you know arrays, cause who doesn't? All methods they provide, functional programming possibilities and more are just so impressive. But, if so, then what **TypedArrays** are and why do we need them?

TypedArrays instead of having a single class on their own, is a name used to reference different types of these specific structures. They basically serve as custom, **array-like views** to **binary data buffers**, which I guess require a bit more explanation. 😊

ArrayBuffer

ArrayBuffer is a class used to contain **fixed-length raw binary data**. 🗄️ You can create one by using its constructor with a length argument, indicating the **number of bytes** for your buffer.

```
const buffer = new ArrayBuffer(8);  
1
```

ArrayBuffers don't have many properties of their own. Most notable being `byteLength` and `slice()` - one for retrieving the length of the buffer in bytes (like the provided one) and other for slicing the specified part of the buffer and creating the new one. The only way you can interact with ArrayBuffers is through so-called **view** - either **TypedArray** or **DataView** (but that's a story for another day).

The importance of ArrayBuffers comes from the way in which they **represent your data** - raw binary. Such form is required by some low-level API, like **WebGL**, because of its **efficiency** ⚡ and integration 🗄️ with other parts of code, like e.g. shaders.

TypedArray[s]

Now, that we know that TypedArrays serve as a view for ArrayBuffer, let's first list 'em all!

- **Int[8/16/32]Array** - for interpreting buffers as arrays of **integer** numbers with the **given number of bits** for representing each;
- **Uint[8/16/32]Array** - **unsigned integer** numbers with the **given number of bits** for each;
- **Float[8/16/32/64]Array** - **floating-point** numbers with the **given number of bits** for each;
- **BigInt64Array** - **integer numbers** (bigint) with **64 bits** for each;
- **BigUint64Array** - **unsigned integer** (bigint) numbers with **64 bits** for each;

Each of the above types of TypedArrays has the **same set** of **methods** and **properties**, with the only difference being in the way of representing the data. TypedArray instance can be created with a given **length**

(creating ArrayBuffer internally), **another TypedArray, an object** (with length and values for given indexes as keys) or previously instantiated **ArrayBuffer**. 📄 📄

Usage

Now, as you have your TypedArray ready, you can freely edit it with methods similar to a normal array. 📄

```
const typedArr = new Uint8Array([0,1,2,3,4]);
const mapped = typedArr.map(num => num * 2);
12
```

One thing to note though, because, as under-the-hood you're operating on the ArrayBuffer's data, your TypedArray has **fixed size**. Furthermore, all methods that can be found in normal arrays, that edit their size (removing, adding, cutting, etc.) have **limited** possibilities or are completely **unavailable**.

```
const typedArr = new Uint8Array([0,1,2,3,4]);
typedArr.push(5)
12
```

You can also iterate on these and **convert** them to **standard arrays** back and forth, whenever you want.

```
const typedArr = new Uint8Array([0,1,2,3,4]);
for(const num of typedArr){

}
const arr = Array.from(typedArr);
12345
```

TypedArrays provide certain functionalities related to its binary-side too! You can e.g. access the underlying ArrayBuffer instance with **buffer** property and read its byte length and offset using **byteLength** and **byteOffset** respectively. 😊

Use-cases

As I mentioned before, **ArrayBuffers** have **big potential** because of the way they represent data. Such compact form can be easily used in many, many places. It can be e.g. **vector** 📐 or other **compressed data** 📦 sent from a server, packed for **maximum speed** and **performance** at all stages - compression, transfer, and decompression. In addition, as I said earlier, some **Web APIs** make good use of the efficiency this format brings. 🎧

With TypedArrays on top of ArrayBuffers, it's so much easier to **manipulate the data** inside (definitely better than setting bits themselves 😊). Beyond one and only limit of fixed size, you can interact with this compact data pretty much the way you would with everyday arrays.

Sets

Continuing our research of array-like structures, we're getting to **Sets**. 📁 These are extremely similar to arrays - they can be used to store data in a similar way, with only one important difference. All of Set's values must be **unique** (there are some weird cases tho 😬) - whether we're talking about **primitive values** or **object references** - doubles are automatically removed.

Usage

Creating Sets is easy - you just need to use the right constructor with an optional argument to provide data from the start.

```
const dataSet = new Set([1, 2, 3, 4, 5]);  
1
```


Sets provide pretty expressive API of their own. Most important being methods like:

- **add()** - appends given value to the end of the Set;
- **delete()** - removes given value from the Set;
- **has()** - checks if given value is present in the Set;
- **clear()** - removes all values from the Set;

They can also be converted to standard arrays and **iterated** at will.


```
const dataSet = new Set([1,2,3]);
const values = [0,1,2,3,4];
for(const value of values) {
  if(dataSet.has(value)){
    dataSet.delete(value)
  } else {
    dataSet.add(value);
  }
}
const result = Array.from(dataSet);
12345678910
```

Use-cases



Most use cases of Sets are clearly based on their ability to store **unique values only**. ⚡ Using such a technique with mere arrays would require some additional boilerplate. Therefore unique values can be especially useful when **storing IDs** and alike. 

Second, **removing elements** in Sets is much more convenient. Just providing the value to delete instead of doing whole find-index-and-splice procedure, is just much more convenient. 🗑️ This, of course, wouldn't be possible so-easily with repetitive values that standard arrays allow.


WeakSets

Now, let's talk about different kind of sets - **WeakSets**.  WeakSets are special - they store values differently, but also have some additional limitations, like much **smaller API**.

Memory

First, a word about how WeakSets store their values. **Only objects** can be used as WeakSets' values. No primitives allowed.  This is very important because of the "*weak*" way in which WeakSets store their data. "Weak" means that if there is **no other reference** to a given object (object are accessed by reference), they can be **garbage-collected**  -

removed at any moment. Thus, a good understanding of references and how objects are interacted with is required to properly utilize the potential of **weak structures**.

Because WeakSets are still... sets, all values they store must be unique. But, as you might know, it's not a big deal with objects - the only possible type of WeakSets' values. As all of them are **stored by  reference**, even objects with exactly the same properties, are considered different.

Usage

API of WeakSets is greatly limited when compared to normal Sets. Probably most important is the fact that they're **not iterable**. They don't have any properties (Sets have e.g. **size** indicating number of values they store) and only 3 major methods - **add()**, **delete()** and **has()**. Constructor method looks the same, only that optional array argument needs to store objects only. However, the use of such an argument **doesn't have much sense**, as all objects you store need to **be referenced** in some other place in your code.

```
const weakDataSet = new WeakSet();
const obj = {a: 10};
weakDataSet.add(obj);
weakDataSet.add({b: 10});
weakDataSet.has(obj);
weakDataSet.has({a: 10});
123456
```

Use-cases

It might be quite hard to find good use-cases for WeakSets actually. That's because, in reality, there aren't many, and they're really specific. The most popular and probably the best one is called **object tagging**. You can use your WeakSets to **group** and thus **tag** specific object when they've been referenced somewhere else in your code. Tagging or grouping as some might like to call it can be a very useful technique if used properly. ⚠

You need to be cautious, however. Remember that all objects that aren't referenced anywhere else, will be **garbage-collected**. But, it doesn't

mean that they'll be removed immediately, but on the **next cycle** 🕒 of the garbage collector. You should keep that fact in mind, and **don't trust WeakSets** too much - some values can be removed sooner or later.

Maps

Maps, IMHO are structures that make the best of both worlds - arrays and object. Inside them, all data is stored in **key-value** pairs. 🗃️ The difference between such method and usual objects can be further noticed in **the API**. What's more, in Maps, keys and values are treated **equally**, meaning you can do even something as creative as setting an object (but remember that you need a reference to it for later access) as an actual key for your value! Also, unlike in objects, pairs stored in Maps have a specific order and are easily **iterable**. ↻

Usage

You can create your Map instance with straight-forward constructor call. You can optionally provide **an array of key-value arrays** upfront as starting values for your Map.

```
const map = new Map([["key1", 10], [10, "value2"]]);  
1
```

It's when it comes to API where Maps really shine. It allows you to make specific operations faster and in a much more readable way.

There's one special property called **size** (available in Sets too) that can give you a quick note about **the number of key-value pairs** at the given moment. What's special about that is the fact that there's no similar, easy enough way to do the same in old-school objects. 😊

And the benefits of this intuitive API don't end here! If you already like the API of Sets, you might be happy to know that it shares many similarities with the API of Maps. All methods used to edit Maps values can feel like modified to new **key-value schema**, methods of Sets. Only the **add()** method has been transformed to **set()** for obvious, rational-thinking-related reasons. 😊 Other than that, to change and access Maps data, you operate mainly with keys instead of values.

Also, just like Sets and objects (it might not be as relevant when it comes to more array-like Sets), Maps provide 3 methods for reading specific groups of their data:

- `entries()` - returns Map's key-value pairs in form of an array of arrays;
- `values()` - returns all of Map's values in an array;
- `keys()` - returns all of Map's keys in an array;

These methods (especially if you're practicing functional programming), were most likely extensively used when interacting with object, as there was no other, convenient way. It shouldn't be the case at all with Maps. With Maps' API and fine **data structure**, you should definitely feel your life being a bit easier. 🌀

```
const map = new Map([['key', 10], ['key2', 10]])
map.forEach((value, key) => {
  map.delete(key);
  map.set(key, 10);
});
```

123456

Use-cases

As you can see, Maps give you a **great alternative** for standard objects. Whenever you need to **access** both key and its value at the same time and be able to **iterate** over them, Maps might be your best option.

This nice combination of iterable and object-like form clearly can has many implementations. And, while you can quite easily create the same effect with a normal object - why bother at all? The convenience behind this **brilliant API** and the fact that it's an industry standard makes Maps a good choice for a lot of different cases. 🌀

WeakMaps

WeakMaps are the second weak structures that we've met. Many facts from WeakSets apply here too! This includes the way of storing data, **object-only** rule, **limited API** and **no iteration** (there's no method giving

you the list of these weakly-stored keys).

As you know, Maps (as well as WeakMaps) store data in the **key-value schema**. This means that there are in fact two collections of data in this one structure - keys and values. The "*weak*" part of WeakMaps applies only to **keys**, because it is them who are responsible for allowing us to access values. Mentioned values are stored in normal or if you like the name, **strong way**. 🤖 So, as weird as it may feel, in WeakMaps, only objects can be used as valid keys.

Usage

Just like with WeakSets, WeakMaps API is severely limited. All methods you can use are `get()`, `set()`, `delete()` and `has()`. Again, **no iteration**. 🤖 But, if you consider the possible use-cases and how such structures work, you'll begin to better understand these limits. You cannot iterate over something that's **weakly stored**. You need references to your keys and so these 4 basic methods are the best way to go. Etc., etc. 😊



Of course, the constructor takes additional, but a not-so-much-useful argument for initiating data.

```
const weakMap = new WeakMap();
const value = {a: 10}
weakMap.set({}, value);
weakMap.set(value, 10)
1234
```

Use-cases

WeakMaps have similar use-cases to WeakSets - **tagging**. All this stuff is happening on the side of keys. Values, however, as **strongly-stored** data of **different types** don't have to be garbage-collected together with the specific key. If saved to a variable earlier, it can still be freely used. This means that you can tag not only one (keys) but also the other side (values) of data and depend on the relations between the two. 🤖

Is that all?

For now - yes.  I hope that this article helped you learn something new or at least remind some basics. Your JS code doesn't have to be dependent only on objects and arrays, especially with modern browsers taking more and more market share.  Also, apart from weak structures and their internal behavior, all structures above have pretty simple and nice **polyfill options**. In this way, you can freely use them, even if it's only for their fine API.