Map, Set, WeakMap and WeakSet

Now we've learned about the following complex data structures:

- Objects for storing keyed collections.
- Arrays for storing ordered collections.

But that's not enough for real life. That's why `Map` and `Set` also exist.

Map

Map is a collection of keyed data items, just like an `Object`. But the main difference is that `Map` allows keys of any type.

The main methods are:

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `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.

For instance:

```
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

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'

alert( map.size ); // 3
```

As we can see, unlike objects, keys are not converted to strings. Any type of key is possible.

Map can also use objects as keys.

For instance:

```
let john = { name: "John" };

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

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

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

Using objects as keys is one of most notable and important `Map` features. For string keys, `Object` can be fine, but it would be difficult to replace the `Map` with a regular `Object` in the example above.

In the old times, before `Map` existed, people added unique identifiers to objects for that:

```
// we add the id field
let john = { name: "John", id: 1 };

let visitsCounts = {};

```

```
// now store the value by id
visitsCounts[john.id] = 123;

alert( visitsCounts[john.id] ); // 123
```

…But `Map` is much more elegant.

How `Map` compares keys

To test values for equivalence, `Map` uses the algorithm SameValueZero. It is roughly the same as strict equality `===`, but the difference is that `NaN` is considered equal to `NaN`. So `NaN` can be used as the key as well.

This algorithm can't be changed or customized.

Chaining

Every `map.set` call returns the map itself, so we can "chain" the calls:

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

Map from Object

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

```
// array of [key, value] pairs
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
}));
```

Here, `Object.entries` returns the array of key/value pairs: `[ ["name","John"], ["age", 30] ]`. That's what `Map` needs.

Iteration over Map

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's used by default in `for..of`.

For instance:

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

// 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); // 500, 350, 50
}
```

```

// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of recipeMap.entries()
  alert(entry); // cucumber,500 (and so on)
}
```

The insertion order is used

The iteration goes in the same order as the values were inserted. `Map` preserves this order, unlike a regular `Object`.

Besides that, `Map` has a built-in `forEach` method, similar to `Array`:

```
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

Set

A `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` – is the elements count.

For example, we have visitors coming, and we'd like to remember everyone. 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 pete = { name: "Pete" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(pete);
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 Pete and Mary)
}
```

The alternative to `Set` could be an array of users, and the code to check for duplicates on every insertion using arr.find. But the performance would be much worse, because this method walks through the whole array checking every element. `Set` is much better optimized internally for uniqueness checks.

Iteration over Set

We can loop over a set either with `for..of` or using `forEach`:

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// the same with forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Note the funny thing. The `forEach` function in the `Set` has 3 arguments: a value, then *again a value*, and then the target object. Indeed, the same value appears in the arguments twice.

That's for compatibility with `Map` where `forEach` has three arguments. Looks a bit strange, for sure. But may help to replace `Map` with `Set` in certain cases with ease, and vice versa.

The same methods `Map` has for iterators are also supported:

- `set.keys()` – returns an iterable object for values,
- `set.values()` – same as `set.keys`, for compatibility with `Map`,
- `set.entries()` – returns an iterable object for entries `[value, value]`, exists for compatibility with `Map`.

WeakMap and WeakSet

`WeakSet` is a special kind of `Set` that does not prevent JavaScript from removing its items from memory. `WeakMap` is the same thing for `Map`.

As we know from the chapter Garbage collection, JavaScript engine stores a value in memory while it is reachable (and can potentially be used).

For instance:

```
let john = { name: "John" };

// the object can be accessed, john is the reference to it

// overwrite the reference
john = null;

// the object will be removed from memory
```

Usually, properties of an object or elements of an array or another data structure are considered reachable and kept in memory while that data structure is in memory.

For instance, if we put an object into an array, then while the array is alive, the object will be alive as well, even if there are no other references to it.

Like this:

```
let john = { name: "John" };

let array = [ john ];

john = null; // overwrite the reference

// john is stored inside the array, so it won't be garbage-collected
// we can get it as array[0]
```

Or, if we use an object as the key in a regular `Map`, then while the `Map` exists, that object exists as well. It occupies memory and may not be garbage collected.

For instance:

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // overwrite the reference

// john is stored inside the map,
// we can get it by using map.keys()
```

`WeakMap/WeakSet` are fundamentally different in this aspect. They do not prevent garbage-collection of key objects.

Let's explain it starting with `WeakMap`.

The first difference from `Map` is that `WeakMap` keys must be objects, not primitive values:

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // works fine (object key)
```

```

// can't use a string as the key
weakMap.set("test", "Whoops"); // Error, because "test" is not an object
```

Now, if we use an object as the key in it, and there are no other references to that object – it will be removed from memory (and from the map) automatically.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // overwrite the reference

// john is removed from memory!
```

Compare it with the regular `Map` example above. Now if `john` only exists as the key of `WeakMap` – it is to be automatically deleted.

`WeakMap` does not support iteration and methods `keys()`, `values()`, `entries()`, so there's no way to get all keys or values from it.

`WeakMap` has only the following methods:

- `weakMap.get(key)`
- `weakMap.set(key, value)`
- `weakMap.delete(key)`
- `weakMap.has(key)`

Why such a limitation? That's for technical reasons. If an object has lost all other references (like `john` in the code above), then it is to be garbage-collected automatically. But technically it's not exactly specified *when the cleanup happens*.

The JavaScript engine decides that. It may choose to perform the memory cleanup immediately or to wait and do the cleaning later when more deletions happen. So, technically the current element count of a `WeakMap` is not known. The engine may have cleaned it up or not, or did it partially. For that reason, methods that access `WeakMap` as a whole are not supported.

Now where do we need such thing?

The idea of `WeakMap` is that we can store something for an object that should exist only while the object exists. But we do not force the object to live by the mere fact that we store something for it.

```
weakMap.set(john, "secret documents");
// if john dies, secret documents will be destroyed automatically
```

That's useful for situations when we have a main storage for the objects somewhere and need to keep additional information, that is only relevant while the object lives.

Let's look at an example.

For instance, we have code that keeps a visit count for each user. The information is stored in a map: a user is the key and the visit count is the value. When a user leaves, we don't want to store their visit count anymore.

One way would be to keep track of users, and when they leave – clean up the map manually:

```
let john = { name: "John" };

// map: user => visits count
let visitsCountMap = new Map();

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

// now john leaves us, we don't need him anymore
john = null;

// but it's still in the map, we need to clean it!
alert( visitsCountMap.size ); // 1
// and john is also in the memory, because Map uses it as the key
```

Another way would be to use `WeakMap`:

```
let john = { name: "John" };

let visitsCountMap = new WeakMap();

visitsCountMap.set(john, 123);

// now john leaves us, we don't need him anymore
john = null;

// there are no references except WeakMap,
// so the object is removed both from the memory and from visitsCountMap automatically
```

With a regular `Map`, cleaning up after a user has left becomes a tedious task: we not only need to remove the user from its main storage (be it a variable or an array), but also need to clean up the additional stores like `visitsCountMap`. And it can become cumbersome in more complex cases when users are managed in one place of the code and the additional structure is in another place and is getting no information about removals.

`WeakMap` can make things simpler, because it is cleaned up automatically. The information in it like visits count in the example above lives only while the key object exists.

`WeakSet` behaves similarly:

- It is analogous to `Set`, but we may only add objects to `WeakSet` (not primitives).
- An object exists in the set while it is reachable from somewhere else.
- Like `Set`, it supports `add`, `has` and `delete`, but not `size`, `keys()` and no iterations.

For instance, we can use it to keep track of whether a message is read:

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];

// fill it with array elements (3 items)
let unreadSet = new WeakSet(messages);

// use unreadSet to see whether a message is unread
alert(unreadSet.has(messages[1])); // true

// remove it from the set after reading
unreadSet.delete(messages[1]); // true

// and when we shift our messages history, the set is cleaned up automatically
messages.shift();

// no need to clean unreadSet, it now has 2 items
// (though technically we don't know for sure when the JS engine clears it)
```

The most notable limitation of `WeakMap` and `WeakSet` is the absence of iterations, and inability to get all current content. That may appear inconvenient, but does not prevent `WeakMap/WeakSet` from doing their main job – be an "additional" storage of data for objects which are stored/managed at another place.

Summary

Regular collections:

- `Map` – is a collection of keyed values.

  The differences from a regular `Object`:

  - Any keys, objects can be keys.
  - Iterates in the insertion order.
  - Additional convenient methods, the `size` property.
- `Set` – is a collection of unique values.
  - Unlike an array, does not allow to reorder elements.
  - Keeps the insertion order.

Collections that allow garbage-collection:

- `WeakMap` – a variant of `Map` that allows only objects as keys and removes them once they become inaccessible by other means.

  - It does not support operations on the structure as a whole: no `size`, no `clear()`, no iterations.
- `WeakSet` – is a variant of `Set` that only stores objects and removes them once they become inaccessible by other means.
  - Also does not support `size/clear()` and iterations.

`WeakMap` and `WeakSet` are used as “secondary” data structures in addition to the “main” object storage. Once the object is removed from the main storage, if it is only found in the `WeakMap/WeakSet`, it will be cleaned up automatically.

Tasks

Filter unique array members

importance: 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 = ["Hare", "Krishna", "Hare", "Krishna",
  "Krishna", "Krishna", "Hare", "Hare", ":-O"
];

alert( unique(values) ); // Hare, Krishna, :-O
```

P.S. Here strings are used, but can be values of any type.

P.P.S. Use `Set` to store unique values.

Open a sandbox with tests.

solution

Open the solution with tests in a sandbox.

Filter anagrams

importance: 4

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.

Open a sandbox with tests.

solution

To find all anagrams, let's split every word to letters and sort them. When letter-sorted, all anagrams are same.

For instance:

```
nap, pan -> anp
ear, era, are -> aer
cheaters, hectares, teachers -> aceehrst
...
```

We'll use the letter-sorted variants as map keys to store only one value per each key:

```
function aclean(arr) {
  let map = new Map();

  for (let word of arr) {
    // split the word by letters, sort them and join back
    let sorted = word.toLowerCase().split('').sort().join(''); // (*)
    map.set(sorted, word);
  }

  return Array.from(map.values());
}

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

alert( aclean(arr) );
```

Letter-sorting is done by the chain of calls in the line `(*)`.

For convenience let's split it into multiple lines:

```
let sorted = arr[i] // PAN
  .toLowerCase() // pan
  .split('') // ['p','a','n']
  .sort() // ['a','n','p']
  .join(''); // anp
```

Two different words `'PAN'` and `'nap'` receive the same letter-sorted form `'anp'`.

The next line put the word into the map:

```
map.set(sorted, word);
```

If we ever meet a word the same letter-sorted form again, then it would overwrite the previous value with the same key in the map. So we'll always have at maximum one word per letter-form.

At the end `Array.from(map.values())` takes an iterable over map values (we don't need keys in the result) and returns an array of them.

Here we could also use a plain object instead of the `Map`, because keys are strings.

That's how the solution can look:

```
function aclean(arr) {
  let obj = {};

  for (let i = 0; i < arr.length; i++) {
    let sorted = arr[i].toLowerCase().split("").sort().join("");
    obj[sorted] = arr[i];
  }

  return Object.values(obj);
}

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

alert( aclean(arr) );
```

Open the solution with tests in a sandbox.

Iterable keys

importance: 5

We want to get an array of `map.keys()` and go on working with it (apart from the map itself).

But there's a problem:

```
let map = new Map();

map.set("name", "John");

let keys = map.keys();

// Error: numbers.push is not a function
keys.push("more");
```

Why? How can we fix the code to make `keys.push` work?

solution

That's because `map.keys()` returns an iterable, but not an array.

We can convert it into an array using `Array.from`:

```
let map = new Map();

map.set("name", "John");

let keys = Array.from(map.keys());

keys.push("more");

alert(keys); // name, more
```

Store "unread" flags

importance: 5

There's an array of messages:

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];
```

Your code can access it, but the messages are managed by someone else's code. New messages are added, old ones are removed regularly by that code, and you don't know the exact moments when it happens.

Now, which data structure you could use to store information whether the message "have been read"? The structure must be well-suited to give the answer "was it read?" for the given message object.

P.S. When a message is removed from `messages`, it should disappear from your structure as well.

P.P.S. We shouldn't modify message objects directly. If they are managed by someone else's code, then adding extra properties to them may have bad consequences.

solution

The sane choice here is a `WeakSet` :

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];

let readMessages = new WeakSet();

// two messages have been read
readMessages.add(messages[0]);
readMessages.add(messages[1]);
// readMessages has 2 elements

// ...let's read the first message again!
readMessages.add(messages[0]);
// readMessages still has 2 unique elements

// answer: was the message[0] read?
alert("Read message 0: " + readMessages.has(messages[0])); // true

messages.shift();
// now readMessages has 1 element (technically memory may be cleaned later)
```

The `WeakSet` allows to store a set of messages and easily check for the existance of a message in it.

It cleans up itself automatically. The tradeoff is that we can't iterate over it. We can't get "all read messages" directly. But we can do it by iterating over all messages and filtering those that are in the set.

P.S. Adding a property of our own to each message may be dangerous if messages are managed by someone else's code, but we can make it a symbol to evade conflicts.

Like this:

```
// the symbolic property is only known to our code
let isRead = Symbol("isRead");
messages[0][isRead] = true;
```

Now even if someone else's code uses `for..in` loop for message properties, our secret flag won't appear.

Store read dates

importance: 5

There's an array of messages as in the previous task. The situation is similar.

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];
```

The question now is: which data structure you'd suggest to store the information: "when the message was read?".

In the previous task we only needed to store the "yes/no" fact. Now we need to store the date and it, once again, should disappear if the message is gone.

solution

To store a date, we can use `WeakMap` :

```
let messages = [
    {text: "Hello", from: "John"},
    {text: "How goes?", from: "John"},
    {text: "See you soon", from: "Alice"}
];

```

```
let readMap = new WeakMap();

readMap.set(messages[0], new Date(2017, 1, 1));
// Date object we'll study later
```

Previous lesson Next lesson

Share Tutorial map

Comments

- You're welcome to post additions, questions to the articles and answers to them.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)
- If you can't understand something in the article – please elaborate.