19th January 2020

Map and Set

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.

Methods and properties 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()` – removes everything from 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[key]` isn't the right way to use a `Map`

Although `map[key]` also works, e.g. we can set `map[key] = 2`, this is treating `map` as a plain JavaScript object, so it implies all corresponding limitations (no object keys and so on).

So we should use `map` methods: `set`, `get` and so on.

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 not for object keys.

Let's try:

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

let visitsCountObj = {}; // try to use an object

visitsCountObj[john] = 123; // try to use john object as the key

// That's what got written!
alert( visitsCountObj["[object Object]"] ); // 123
```

As `visitsCountObj` is an object, it converts all keys, such as `john` to strings, so we've got the string key `"[object Object]"`. Definitely not what we want.

How `Map` compares keys

To test keys 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');
```

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`:

```
// runs the function for each (key, value) pair
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

Object.entries: Map from Object

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

```
// array of [key, value] pairs
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

If we have a plain object, and we'd like to create a `Map` from it, then we can use built-in method Object.entries(obj) that returns an array of key/value pairs for an object exactly in that format.

So we can create a map from an object like this:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

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

Object.fromEntries: Object from Map

We've just seen how to create `Map` from a plain object with `Object.entries(obj)`.

There's `Object.fromEntries` method that does the reverse: given an array of `[key, value]` pairs, it creates an object from them:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

We can use `Object.fromEntries` to get an plain object from `Map`.

E.g. we store the data in a `Map`, but we need to pass it to a 3rd-party code that expects a plain object.

Here we go:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries()); // make a plain object (*)

// done!
// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2
```

A call to `map.entries()` returns an array of key/value pairs, exactly in the right format for `Object.fromEntries`.

We could also make line `(*)` shorter:

```
let obj = Object.fromEntries(map); // omit .entries()
```

That's the same, because `Object.fromEntries` expects an iterable object as the argument. Not necessarily an array. And the standard iteration for `map` returns same key/value pairs as `map.entries()`. So we get a plain object with same key/values as the `map`.

Set

A `Set` is a special type collection – "set of values" (without keys), where each value may occur only once.

Its main methods are:

- `new Set(iterable)` – creates the set, and if an `iterable` object is provided (usually an array), copies values from it into the set.
- `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.

The main feature is that repeated calls of `set.add(value)` with the same value don't do anything. That's the reason why each value appears in a `Set` only once.

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 callback function passed in `forEach` has 3 arguments: a `value`, then *the same value* `valueAgain`, and then the target object. Indeed, the same value appears in the arguments twice.

That's for compatibility with `Map` where the callback passed `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`.

Summary

`Map` – is a collection of keyed values.

Methods and properties:

- `new Map([iterable])` – creates the map, with optional `iterable` (e.g. array) of `[key,value]` pairs for initialization.
- `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()` – removes everything from the map.
- `map.size` – returns the current element count.

The differences from a regular `Object`:

- Any keys, objects can be keys.
- Additional convenient methods, the `size` property.

`Set` – is a collection of unique values.

Methods and properties:

- `new Set([iterable])` – creates the set, with optional `iterable` (e.g. array) of values for initialization.
- `set.add(value)` – adds a value (does nothing if `value` exists), 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.

Iteration over `Map` and `Set` is always in the insertion order, so we can't say that these collections are unordered, but we can't reorder elements or directly get an element by its number.

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"
];
```

```
8
9  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

Filter anagrams

importance: 4

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

For instance:

```
1  nap - pan
2  ear - are - era
3  cheaters - hectares - teachers
```

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

For instance:

```
1  let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];
2
3  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

Iterable keys

importance: 5

We'd like to get an array of `map.keys()` in a variable and then do apply array-specific methods to it, e.g. `.push`.

But that doesn't work:

```
let map = new Map();

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

let keys = map.keys();

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

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

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- 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…)