

Map and set, weakmap and weakset

Map

Very similar to an object, however, with object the only key that is allowed is a String. Map on the other hand allows keys of any type. Any type as it ANY type, even an object can be used as a key

- `new Map()`: Creates a new empty map
- `map.set(key, value)`: Sets the key-value pair into the map
- `map.get(key)`: Returns the value by the `key`, returns `undefined` if `key` doesn't exist in map
- `map.has(key)`: Returns `true` if the `key` exists, `false` otherwise
- `map.delete(key)`: Removes the key-value pair by the `key`
- `map.clear()`: Removes everything from the map
- `map.size`: Returns the total number of key-value pair

The way that `Map` compares keys is roughly the same as `===`, but `NaN` is considered to be equal to `NaN` hence, even `NaN` can be used as key as well!

Iteration over Map

Three ways of iterating over a map

1. `map.keys()`: Returns an iterable for keys
2. `map.values()`: Returns an iterable for values
3. `map.entries()`: Returns an iterable for both key and value, this is the default for `for ... of`

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

for (let vegs of recipeMap.keys()) {
  []console.log(vergs); // cucumber, tomatoes, onion
}

for (let amount of recipeMap.values()) {
```

```
    console.log(amount); //500, 350, 50
  }

  for (let entry of recipeMap) {
    console.log(entry) // [cucumber, 500], [tomatoes, 350], [onion, 50]
  }
```

The iteration follows the insertion order, unlike object which doesn't preserve the insertion order in iteration.

Map from array

You can create a map from a 2D array like below:

```
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, true]
])
```

Map from object

You can create a map from an object like below:

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

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

`Object.entries(obj)` will return a 2D array where the 1D array will be the two key-value properties. Since all of the key in object are String the key will always be a String.

Object from Map

`Object.fromEntries` does the opposite, it will create an object from a map. All of the key from the map will be converted to a String, because keep in mind that object can only take String as it's key, nothing else. The values will be kept as the same.

```
let map3 = new Map();
map3.set(1, "50");
map3.set("name", "Ricky");
```

```
map3.set("2", "Ricky");
```

```
let obj = Object.fromEntries(map3); // {"1": "50", "name": "Ricky", "2": "Ricky"}.
```

Set

A set of unique values. There is no key-value pair mapping, `Set` only contains the values, and the same value may only occur once.

- `new Set([iterable])`: To create a set, if the `iterable` object is provided, it creates a set from those values
- `set.add(value)`: Add value to the set, and return the set itself
- `set.delete(value)`: Remove the value, returns `true` if `value` existed otherwise `false`
- `set.has(value)`: `true` if it contains the `value` `false` otherwise.
- `set.clear()`: Removes everything from the set
- `set.size`: Returns the number of elements in the set

Set iteration

You can iterate over a set using same `for ... of` loop.

1. `set.keys()`: Return the iterable object for values
2. `set.values()`: This is the same as `set.keys()`
3. `set.entries()`: Return iterable object with entries `[value, value]`

These method exists in order to be compatible with `Map` if you decide to switch from one to the other.

WeakMap

With normal `Map` the object that is mapped as the value will be kept in memory and so long as the `Map` exists, the object will exist as well.

`WeakMap` on the other hand is different in handling how the garbage collection work. It doesn't prevent garbage-collection of key objects like `Map` does, hence their use cases are completely different.

Here is how `WeakMap` works.

```
let weakmap = new WeakMap();
```

```
let obj = {};
```

```
weakmap.set(obj, "ok");
```

First of all, `WeakMap` can only take object as the key, if you try to use a primitive like String or integer it will result in an error.

Then, if the object we used as the key have no other references to that object (excluding the one from `WeakMap`) it will be removed from memory automatically.

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

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

john = null; // the object now lost the reference!

// john is now lost from memory, because WeakMap doesn't prevent garbage collection of key
objects.
```

In addition, there are limited functionality to `WeakMap`, there is no iteration, no way to get all keys or values from it.

It only supports `set`, `get`, `delete`, `has` method like a normal map.

Use cases

One of the use cases for `WeakMap` is for caching. The result from a function call associated with an object can be stored in a `WeakMap`, future calls on the same object can reuse the same result. Then when you want to clean up the `cache`, you can just delete the reference to the object and that result in `WeakMap` cache will be automatically removed from memory since it gets garbage collected.

WeakSet

Behaves similarly, but you can only add objects to `WeakSet` no primitives. Again only supports `add`, `has`, `delete` no way of doing iterations.

Used for a yes/no fact, say if an object is still being used somewhere else or not.

The object will also be removed from the `WeakSet` once the object becomes inaccessible/unreachable.