Keyed collections

This chapter introduces collections of data which are indexed by a key; `Map` and `Set` objects contain elements which are iterable in the order of insertion.

Maps

Map object

A `Map` object is a simple key/value map and can iterate its elements in insertion order.

The following code shows some basic operations with a `Map`. See also the `Map` reference page for more examples and the complete API. You can use a `for...of` loop to return an array of `[key, value]` for each iteration.

JS

```
const sayings = new Map();
sayings.set("dog", "woof");
sayings.set("cat", "meow");
sayings.set("elephant", "toot");
sayings.size; // 3
sayings.get("dog"); // woof
sayings.get("fox"); // undefined
sayings.has("bird"); // false
sayings.delete("dog");
sayings.has("dog"); // false

for (const [key, value] of sayings) {
  console.log(`${key} goes ${value}`);
}
// "cat goes meow"
// "elephant goes toot"

sayings.clear();
sayings.size; // 0
```

Object and Map compared

Traditionally, objects have been used to map strings to values. Objects allow you to set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. `Map` objects, however, have a few more advantages that make them better maps.

- The keys of an `Object` are strings or symbols, whereas they can be of any value for a `Map`.
- You can get the `size` of a `Map` easily, while you have to manually keep track of size for an `Object`.
- The iteration of maps is in insertion order of the elements.
- An `Object` has a prototype, so there are default keys in the map. (This can be bypassed using `map = Object.create(null)`.)

These three tips can help you to decide whether to use a `Map` or an `Object`:

- Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.
- Use objects when there is logic that operates on individual elements.

WeakMap object

A `WeakMap` is a collection of key/value pairs whose keys must be objects or non-registered symbols, with values of any arbitrary JavaScript type, and which does not create strong references to its keys. That is, an object's presence as a key in a `WeakMap` does not prevent the object from being garbage collected. Once an object used as a key has been collected, its corresponding values in any `WeakMap` become candidates for garbage collection as well — as long as they aren't strongly referred to elsewhere. The only primitive type that can be used as a `WeakMap` key is symbol — more specifically, non-registered symbols — because non-registered symbols are guaranteed to be unique and cannot be re-created.

The `WeakMap` API is essentially the same as the `Map` API. However, a `WeakMap` doesn't allow observing the liveness of its keys, which is why it doesn't allow enumeration. So there is no method to obtain a list of the keys in a `WeakMap`. If there were, the list would depend on the state of garbage collection, introducing non-determinism.

For more information and example code, see also "Why WeakMap?" on the `WeakMap` reference page.

One use case of `WeakMap` objects is to store private data for an object, or to hide implementation details. The following example is from Nick Fitzgerald's blog post "Hiding Implementation Details with ECMAScript 6 WeakMaps" . The private data and methods belong inside the object and are stored in the `privates` object, which is a `WeakMap` . Everything exposed on the instance and prototype is public; everything else is inaccessible from the outside world because `privates` is not exported from the module.

JS

```
const privates = new WeakMap();

function Public() {
  const me = {
    // Private data goes here
  };
  privates.set(this, me);
}

Public.prototype.method = function () {
  const me = privates.get(this);
  // Do stuff with private data in `me`
  // …
};

module.exports = Public;
```

Sets

Set object

`Set` objects are collections of unique values. You can iterate its elements in insertion order. A value in a `Set` may only occur once; it is unique in the `Set` 's collection.

The following code shows some basic operations with a `Set` . See also the `Set` reference page for more examples and the complete API.

JS

```
const mySet = new Set();
mySet.add(1);
mySet.add("some text");
mySet.add("foo");

mySet.has(1); // true
mySet.delete("foo");
```

```
mySet.size; // 2

for (const item of mySet) {
  console.log(item);
}
// 1
// "some text"
```

Converting between Array and Set

You can create an `Array` from a Set using `Array.from` or the spread syntax. Also, the `Set` constructor accepts an `Array` to convert in the other direction.

> **Note:** `Set` objects store *unique values*—so any duplicate elements from an Array are deleted when converting!

JS

```
Array.from(mySet);
[...mySet2];

mySet2 = new Set([1, 2, 3, 4]);
```

Array and Set compared

Traditionally, a set of elements has been stored in arrays in JavaScript in a lot of situations. The `Set` object, however, has some advantages:

- Deleting Array elements by value (`arr.splice(arr.indexOf(val), 1)`) is very slow.
- `Set` objects let you delete elements by their value. With an array, you would have to `splice` based on an element's index.
- The value `NaN` cannot be found with `indexOf` in an array.
- `Set` objects store unique values. You don't have to manually keep track of duplicates.

WeakSet object

`WeakSet` objects are collections of garbage-collectable values, including objects and non-registered symbols. A value in the `WeakSet` may only occur once. It is unique in the `WeakSet` 's collection.

The main differences to the `Set` object are:

- In contrast to `Sets`, `WeakSets` are **collections of *objects or symbols only***, and not of arbitrary values of any type.
- The `WeakSet` is *weak*: References to objects in the collection are held weakly. If there is no other reference to an object stored in the `WeakSet`, they can be garbage collected. That also means that there is no list of current objects stored in the collection.
- `WeakSets` are not enumerable.

The use cases of `WeakSet` objects are limited. They will not leak memory, so it can be safe to use DOM elements as a key and mark them for tracking purposes, for example.

Key and value equality of Map and Set

Both the key equality of `Map` objects and the value equality of `Set` objects are based on the SameValueZero algorithm:

- Equality works like the identity comparison operator `===`.
- `-0` and `+0` are considered equal.
- `NaN` is considered equal to itself (contrary to `===`).