Keyed collections

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

Maps

Map object

In JavaScript, a Map object is a collection of key-value pairs where any value (both objects and primitive values) may be used as either a key or a value. This means that, unlike an array, **keys are not restricted to numbers or strings, they can be any type of value**.

Map objects are similar to Objects but have some key differences. Maps allow any value to be used as a key, whereas objects only allow strings or symbols to be used as keys. Also, the size of a Map object can be easily retrieved using the size property, while with Objects, you need to manually keep track of the number of properties.

To create a Map object, you can use the Map() constructor like this:

```
const myMap = new Map();
```

You can also initialize a Map object with an array of key-value pairs like this:

```
const myMap = new Map([
  ['key1', 'value1'],
  ['key2', 'value2'],
  ['key3', 'value3']
]);
```

Here, `key1`, `key2`, and `key3` are the keys, and `value1`, `value2`, and `value3` are the corresponding values.

Map objects provide several methods to manipulate and retrieve data from the Map. Here are some commonly used methods:

- `set(key, value)` - sets the value for the specified key in the Map object.
- `get(key)` - returns the value associated with the specified key in the Map object, or `undefined` if the key does not exist.
- `has(key)` - returns a Boolean indicating whether the specified key exists in the Map object.
- `delete(key)` - removes the key-value pair associated with the specified key from the Map object.
- `clear()` - removes all key-value pairs from the Map object.
- `size` - returns the number of key-value pairs in the Map object.

Here's an example of how you can use some of these methods:

```
const myMap = new Map();
myMap.set('key1', 'value1');
myMap.set('key2', 'value2');
myMap.set('key3', 'value3');

console.log(myMap.get('key1')); // Output: "value1"
console.log(myMap.has('key2')); // Output: true

myMap.delete('key3');
console.log(myMap); // Output: Map(2) { 'key1' => 'value1', 'key2' => 'value2' }

console.log(myMap.size); // Output: 2

myMap.clear();
console.log(myMap); // Output: Map(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`, where 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 the 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 object is another type of Map in JavaScript, but with a key difference: **the keys in a WeakMap must be objects, and these objects are weakly held**. This means that if there are no other references to a key object, it can be garbage-collected by the JavaScript engine even if it is still in the WeakMap.

In other words, a WeakMap allows you to associate values with objects without creating strong references to those objects. This can be useful in situations where you want to store some data that is associated with an object, but you don't want to prevent that object from being garbage-collected if there are no other references to it.

To create a WeakMap object, you can use the WeakMap() constructor, like this:

```
const myWeakMap = new WeakMap();
```

You can then add key-value pairs to the WeakMap object using the `set()` method, like this:

```
const keyObject = {};
myWeakMap.set(keyObject, "value");
```

Here, `keyObject` is an object that acts as the key in the WeakMap, and `"value"` is the value associated with that key.

Some key differences between a WeakMap and a regular Map are:

- Only objects can be used as keys in a WeakMap.
- WeakMaps are not iterable (i.e., you cannot use a `for...of` loop to iterate over the key-value pairs).

- WeakMaps do not have a `size` property or a `clear()` method.

Some of the methods available to WeakMap objects include `set()`, `get()`, `has()`, and `delete()`. These methods allow you to add, retrieve, check for existence, and remove key-value pairs in the WeakMap.

Here's an example of how you can use a WeakMap object:

```
const myWeakMap = new WeakMap();
const keyObject = {};

myWeakMap.set(keyObject, "value");

console.log(myWeakMap.get(keyObject)); // Output: "value"

keyObject = null; // Removing the only reference to keyObject
console.log(myWeakMap.get(keyObject));  //  Output:  undefined  (keyObject  has  been
                                            garbage-collected)
```

Sets

Set object

A Set object is a collection of unique values in JavaScript (an **array-like object without keys, and where every value is unique and is not indexed like in an array**). It can store any type of value, including primitives (like numbers, strings, and booleans) and object references. However, each value can only occur once in the Set object.

To create a Set object, you can use the `Set()` constructor, like this:.

```
const mySet = new Set();
```

You can then add values to the Set object using the add() method, like this:

```
mySet.add(1);
mySet.add("hello");
mySet.add(true);
```

Here, we've added the number `1`, the string "`hello`", and the boolean `true` to the Set object.

Some methods available to Set objects include `add()`, `delete()`, `has()`, and `clear()`. These methods allow you to add, remove, check for existence, and clear values from the Set object.

Here's an example of how you can use a Set object:

```
const mySet = new Set();

mySet.add(1);
mySet.add(2);
mySet.add(3);

console.log(mySet.has(2)); // Output: true

mySet.delete(2);

console.log(mySet); // Output: Set { 1, 3 }

mySet.clear();

console.log(mySet); // Output: Set(0) {}
```

In this example, we create a Set object and add the numbers `1`, `2`, and `3` to it. We then use the `has()` method to check if the value `2` exists in the Set object (which it does), and use the `delete()` method to remove the value `2`. Finally, we use the `clear()` method to remove all values from the Set object.

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.

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

```
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 objects**. An object in the `WeakSet` may only occur once. It is unique in the `WeakSet`'s collection, and objects are not enumerable.
The main differences to the `Set` object are:

- In contrast to `Sets`, `WeakSets` are **collections of *objects 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 `===`).