Chapter 43: Set

Parameter	Details
iterable	If an iterable object is passed, all of its elements will be added to the new Set. null is treated as undefined.
value	The value of the element to add to the Set object.
callback	Function to execute for each element.
thisArg	Optional. Value to use as this when executing callback.

The Set object lets you store unique values of any type, whether primitive values or object references.

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set may only occur **ONCE**; it is unique in the Set's collection. Distinct values are discriminated using the *SameValueZero* comparison algorithm.

Standard Specification About Set

Section 43.1: Creating a Set

The Set object lets you store unique values of any type, whether primitive values or object references.

You can push items into a set and iterate them similar to a plain JavaScript array, but unlike array, you cannot add a value to a Set if the value already exist in it.

To create a new set:

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

Or you can create a set from any iterable object to give it starting values:

```
const arr = [1,2,3,4,4,5];
const mySet = new Set(arr);
```

In the example above the set content would be `{1, 2, 3, 4, 5}`. Note that the value 4 appears only once, unlike in the original array used to create it.

Section 43.2: Adding a value to a Set

To add a value to a Set, use the `.add()` method:

```
mySet.add(5);
```

If the value already exist in the set it will not be added again, as Sets contain unique values.

Note that the `.add()` method returns the set itself, so you can chain add calls together:

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

Section 43.3: Removing value from a set

To remove a value from a set, use `.delete()` method:

```
mySet.delete(some_val);
```

This function will return `true` if the value existed in the set and was removed, or `false` otherwise.

Section 43.4: Checking if a value exist in a set

To check if a given value exists in a set, use `.has()` method:

```
mySet.has(someVal);
```

Will return `true` if `someVal` appears in the set, `false` otherwise.

Section 43.5: Clearing a Set

You can remove all the elements in a set using the `.clear()` method:

```
mySet.clear();
```

Section 43.6: Getting set length

You can get the number of elements inside the set using the `.size` property

```
const mySet = new Set([1, 2, 2, 3]);
mySet.add(4);
mySet.size; // 4
```

This property, unlike `Array.prototype.length`, is read-only, which means that you can't change it by assigning something to it:

```
mySet.size = 5;
mySet.size; // 4
```

In strict mode it even throws an error:

```
TypeError: Cannot set property size of #<Set> which has only a getter
```

Section 43.7: Converting Sets to arrays

Sometimes you may need to convert a Set to an array, for example to be able to use `Array.prototype` methods like `.filter()`. In order to do so, use `Array.from()` or `destructuring-assignment`:

```
var mySet = new Set([1, 2, 3, 4]);
//use Array.from
const myArray = Array.from(mySet);
//use destructuring-assignment
const myArray = [...mySet];
```

Now you can filter the array to contain only even numbers and convert it back to Set using Set constructor:

```
mySet = new Set(myArray.filter(x => x % 2 === 0));
```

`mySet` now contains only even numbers:

```
console.log(mySet); // Set {2, 4}
```

Section 43.8: Intersection and difference in Sets

There are no build-in methods for intersection and difference in Sets, but you can still achieve that but converting them to arrays, filtering, and converting back to Sets:

```
var set1 = new Set([1, 2, 3, 4]),
    set2 = new Set([3, 4, 5, 6]);

const intersection = new Set(Array.from(set1).filter(x => set2.has(x)));//Set {3, 4}
const difference = new Set(Array.from(set1).filter(x => !set2.has(x))); //Set {1, 2}
```

Section 43.9: Iterating Sets

You can use a simple for-of loop to iterate a Set:

```
const mySet = new Set([1, 2, 3]);

for (const value of mySet) {
  console.log(value); // logs 1, 2 and 3
}
```

When iterating over a set, it will always return values in the order they were first added to the set. For example:

```
const set = new Set([4, 5, 6])
set.add(10)
set.add(5) //5 already exists in the set
Array.from(set) //[4, 5, 6, 10]
```

There's also a `.forEach()` method, similar to `Array.prototype.forEach()`. It has two parameters, `callback`, which will be executed for each element, and optional `thisArg`, which will be used as `this` when executing `callback`.

`callback` has three arguments. The first two arguments are both the current element of Set (for consistency with `Array.prototype.forEach()` and `Map.prototype.forEach()`) and the third argument is the Set itself.

```
mySet.forEach((value, value2, set) => console.log(value)); // logs 1, 2 and 3
```