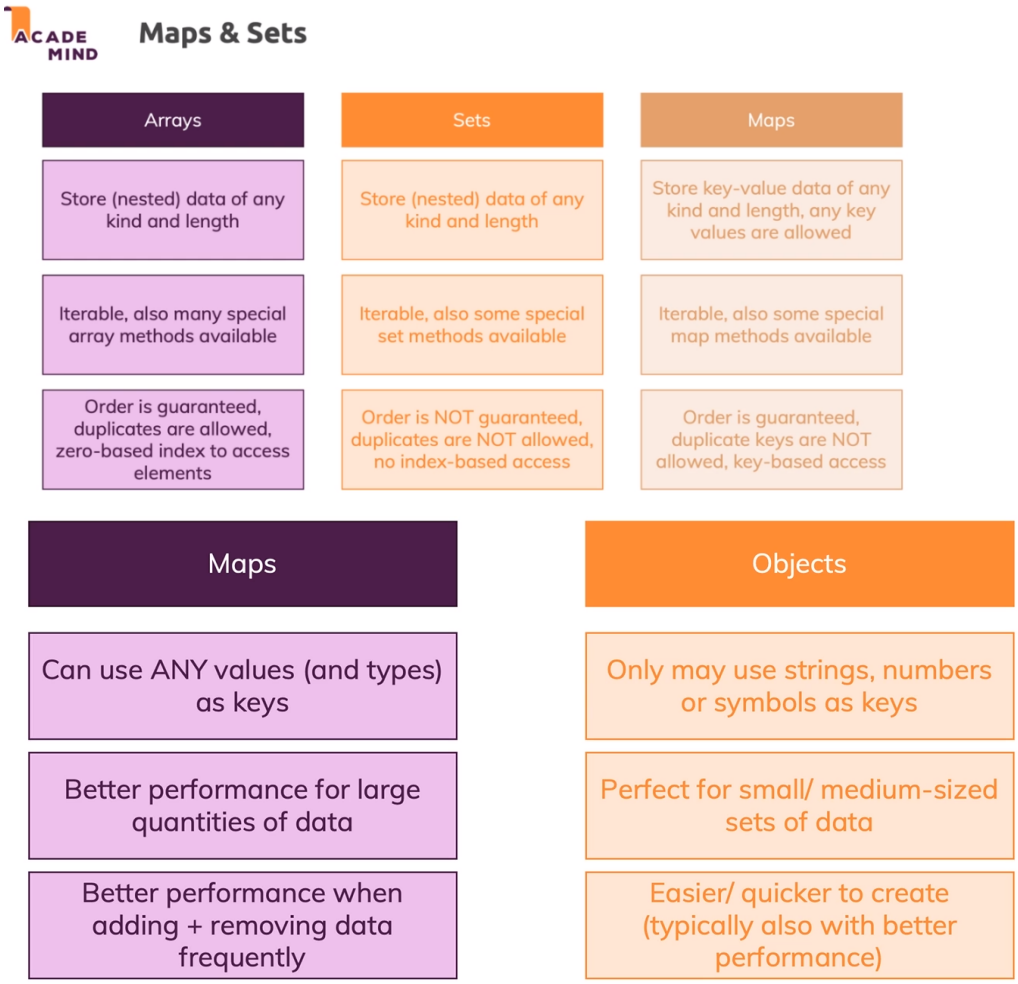
[3] SET AND MAPS



In regards to garbage collection..

- Memory allocation is done automatically by JS
- As long as reference exist, there will not be any cleaning or garbage collecting
- If a location is unreachable then that will be collected/released as garbage.

SET

DM: Set - JavaScript | MDN (mozilla.org)
JavaScript Sets (w3schools.com)

A JavaScript Set is a collection of unique values. Each value can only occur once in a Set. A Set can hold any value of any data type. You can iterate through the elements of a set in insertion order. There is no index for the elements, as they must be unique.

```
---Syntax:
const letters = new Set(["a","b","c"]);

---Operations:
new Set(iterable) – creates the set, and if an iterable object is provided
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.

---Example[1]:
let vegetables = new Set();
let cucumber = { name: "cucumber" };
let onion = { name: "onion" };
let potato = { name: "Potato" };
// add multiple times
vegetables.add(cucumber);
vegetables.add(cucumber);
vegetables.add(onion);
vegetables.add(potato);
vegetables.add(onion);
vegetables.add(potato);
// set is a unique values collectionn
console.log( vegetables.size ); //output: 3
```

Weak Set

In JavaScript, a WeakSet is a built-in object that allows you to store a collection of unique objects. The primary difference between a WeakSet and a regular Set is how they handle references to objects, which affects memory management.
Here are some key characteristics and concepts associated with WeakSets:

1. Uniqueness: Like Sets, WeakSets only store unique values. This means that you can't have duplicate objects in a WeakSet. Each object can only be added to a WeakSet once.

2. Weak References: The most significant difference between WeakSets and Sets is how they hold references to objects. In a regular Set, references to objects are strong, meaning that as long as the object is in the Set, it won't be garbage collected even if it's no longer used in your program. This can lead to memory leaks if you forget to remove objects from the Set when you're done with them. In contrast, a WeakSet holds weak references to objects. This means that if there are no other references to an object outside of the WeakSet, it can be garbage collected even if it's still in the WeakSet. This makes WeakSets particularly useful for scenarios where you want to associate data with objects temporarily and don't want to prevent those objects from being garbage collected when they are no longer needed.
3. Methods: WeakSets have methods similar to Sets for managing their contents. These methods include add(), delete(), and has(). You can use add() to add an object to the WeakSet, delete() to remove an object, and has() to check if an object is in the WeakSet.

Here's an example of using a WeakSet in JavaScript:

```
// Creating a WeakSet
let myWeakSet = new WeakSet();

// Creating some objects
let obj1 = {};
let obj2 = {};
let obj3 = {};

// Adding objects to the WeakSet
myWeakSet.add(obj1);
myWeakSet.add(obj2);

// Checking if objects are in the WeakSet
console.log(myWeakSet.has(obj1)); // true
console.log(myWeakSet.has(obj2)); // true
console.log(myWeakSet.has(obj3)); // false

// Removing an object from the WeakSet
myWeakSet.delete(obj1);

// Checking again
console.log(myWeakSet.has(obj1)); // false
```

Remember that the key feature of WeakSets is that they allow objects to be garbage collected when there are no other references to them, making them suitable for

scenarios where you want to associate data with objects without preventing those objects from being cleaned up by the garbage collector when they are no longer needed.

Maps

DM: [Map - JavaScript | MDN (mozilla.org)]

[JavaScript Maps (w3schools.com)]

`Map` objects are collections of key-value pairs. A key in the `Map` may only occur once; it is unique in the `Map`'s collection. A Map's keys can be any value (including functions, objects, or any primitive), in the case of objects, must be either a String or a Symbol. A Map holds key-value pairs where the keys can be any datatype. A Map remembers the original insertion order of the keys.

```
---Create a Map
const fruits = new Map([
["apples", 500],
["bananas", 300],
["oranges", 200]
]);

---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.

--Example[1]
const foodMap = new Map ([
['burger', 50],
['hotdog', 30],
['pizza', 70]
])
// iterate over keys (foods)
for (let foods of foodMap.keys()) {
console.log(foods);
}
// iterate over values (amounts)
for (let amount of foodMap.values()) {
console.log(amount);
```

```
}
// iterate over [key, value] entries
for (let entry of foodMap) {
console.log(entry);
}
// Create a map from an object like this.
let srcObject = {
name: "John Snow",
title: "King in the North"
};
let map = new Map(Object.entries(srcObject));
console.log( map.get('title') );
```

Weak Maps

In JavaScript, a WeakMap is a built-in object that allows you to create a mapping of keys to values, where the keys are objects and the values can be any JavaScript value. WeakMaps are similar to regular Maps, but with some important differences, primarily related to how they handle references to keys and memory management.
Here are the key characteristics and concepts associated with WeakMaps:

1. Keys Must Be Objects: In a WeakMap, the keys must be objects. Unlike regular Maps where keys can be any data type, WeakMaps only accept objects as keys. This restriction is because WeakMaps use the concept of weak references for keys.
2. Weak References: The most significant difference between WeakMaps and regular Maps is how they hold references to keys. In a regular Map, the references to keys are strong, which means that as long as the key is in the Map, it prevents the associated object from being garbage collected even if there are no other references to that object. In contrast, a WeakMap holds weak references to keys. This means that if there are no other references to a key object outside of the WeakMap, it can be garbage collected. This feature can be especially useful for scenarios where you want to associate data with objects but don't want to prevent those objects from being cleaned up by the garbage collector when they are no longer needed.
3. Methods: WeakMaps have methods similar to Maps for managing their key-value pairs. These methods include set(), get(), delete(), and has(). You can use set() to associate a value with a key, get() to retrieve the value associated with a key, delete() to remove a key-value pair, and has() to check if a key exists in the WeakMap.

Here's an example of using a WeakMap in JavaScript:

```
// Creating a WeakMap
let myWeakMap = new WeakMap();

// Creating some objects
let obj1 = {};
let obj2 = {};

// Associating values with objects in the WeakMap
myWeakMap.set(obj1, 'Value for obj1');
myWeakMap.set(obj2, 'Value for obj2');

// Retrieving values
console.log(myWeakMap.get(obj1)); // 'Value for obj1'
console.log(myWeakMap.get(obj2)); // 'Value for obj2'

// Checking if keys exist in the WeakMap
console.log(myWeakMap.has(obj1)); // true
console.log(myWeakMap.has(obj2)); // true

// Removing a key-value pair
myWeakMap.delete(obj1);

// Checking again
console.log(myWeakMap.has(obj1)); // false
```

Remember that the main advantage of WeakMaps is that they allow objects to be garbage collected when there are no other references to them, making them suitable for scenarios where you want to associate data with objects without preventing those objects from being cleaned up by the garbage collector when they are no longer needed.