ES6 Collections

Before ES6, JavaScript had limited options for working with collections. Arrays and objects were the primary tools:

- **Arrays** stored ordered lists of items, but were limited when it came to searching, modifying, or ensuring uniqueness.
- **Objects** acted as hashmaps, associating string keys with values, but with significant limitations like coercion of non-string keys and performance issues with large datasets.

ES6 (ECMAScript 6) introduced Collections in JavaScript, providing built-in data structures to store and manage data in more flexible ways. The main collections introduced in ES6 are Map, Set, WeakMap, and WeakSet.

- These collections offering new, more powerful ways to handle collections of data.
- These new structures offer performance benefits, better key-value management, and automatic memory management features (in the case of WeakMap & WeakSet).
- These collections brought new data structures to JavaScript, offering more efficient and flexible ways to handle data than traditional objects and arrays.

Key Collection Types

- **Arrays:** Ordered list of elements, where each element is accessible via its index.
  Common Operations: Adding, removing, sorting, filtering, mapping, reducing, etc.
- **Objects:** Collections of key-value pairs where keys are strings or symbols, with common operations like creating, accessing, updating, deleting properties, etc.
  Common Operations: Object.keys(), Object.values(), Object.entries(), etc.
- **Sets:** A collection of unique values, where duplicate elements are automatically removed.
  Common Operations: Adding, deleting, checking for existence, etc.
- **Maps:** A collection of key-value pairs where keys can be any type, not limited to strings.
  Common Operations: Adding, retrieving, deleting key-value pairs.
- **WeakMaps:** Similar to Maps, but keys must be objects, and entries are garbage-collected if the object is no longer referenced.
- **WeakSets:** A collection of objects (no primitive values), where object references are weakly held (garbage-collected if not referenced elsewhere).

Map: A Flexible Key-Value Collection

- The Map object in ES6 addresses several shortcomings of using plain objects as key-value stores.
- In objects, only strings (or symbols) can serve as keys, which means other types (like numbers or objects) get coerced into strings. This can lead to unexpected behavior when keys aren't unique or when using non-string values.

Why is Map Better than Objects?

- **Key Flexibility:** Unlike objects, which restrict keys to strings or symbols, Map allows any data type as a key (strings, numbers, objects, functions, etc.).
- **Order Preservation:** Map maintains the insertion order of key-value pairs, ensuring predictable iteration.
- **Performance:** Operations like searching for a key (get), inserting a key-value pair (set), or deleting a key (delete) are faster and optimized in Map, especially when handling large datasets.
- **Clear Semantics:** Unlike objects, where properties can overlap with prototype methods (e.g., toString), Map does not inherit from Object.prototype, making it a cleaner key-value store.

Internal Structure of Map

- Map internally uses hash maps to store key-value pairs.
- Hash maps provide an efficient lookup mechanism (O(1) on average) by converting keys into hashes (unique fixed-length values).
- This ensures faster lookup and deletion compared to traditional objects, which can have O(n) complexity for large data.

Key Features of Map

- **Unique Keys:** A key in a Map must be unique.
- **Any Data Type for Keys:** Unlike regular objects, where keys are typically strings, Map keys can be of any type (including objects and functions).
- **Maintains Insertion Order:** Maps keep track of the order of key-value pairs.
- **Easy Size Calculation:** You can easily find out how many entries exist using the size property.

Use Cases of Map

- **Storing Unique Data:** When you need to ensure that keys are unique, Map is a better choice than plain objects.
- **Efficient Key-Based Lookup:** When frequent lookups by key are necessary, a Map is ideal.
- **Maintaining Order of Insertion:** If you need to maintain the insertion order, unlike regular objects where the order is unpredictable.
- **Using Objects as Keys:** When objects or other non-string values need to be used as keys, Map is more flexible than objects.
- **Caching Data:** Map can be used to cache expensive calculations or database lookups using keys for quick access.

Basic Syntax

```
const map = new Map();

// Adding entries
        map.set(key, value);
// Accessing entries
        map.get(key);
// Checking existence
        map.has(key);
// Removing entries
        map.delete(key);
// Clearing the map
        map.clear();
// Getting the size
        map.size;
```

Example

```
const map = new Map();
map.set('name', 'Neha Rathore');
map.set('age', 22);
console.log(map.get('name'));
```

Example: Using Objects as Keys

```
const obj = { id: 1 };
const map = new Map();
map.set(obj, 'Value associated with key-object');
console.log(map.get(obj));
```

Example: Iterating Over Map Entries

```
const map = new Map([['name', 'Neha Rathore'], ['age', 22]]);
for (let [key, value] of map) {
  console.log(key, value);
}
```

Example: Using Functions as Keys

```
const map = new Map();
const funcKey = function() {};
map.set(funcKey, 'Here is the function');
console.log(map.get(funcKey));
```

Example: Checking If Key Exists

```
const map = new Map();
map.set('name', 'Neha Rathore');
console.log(map.has('name'));
console.log(map.has('age'));
```

Example: Removing a Key-Value Pair

```
const map = new Map();
map.set('city', 'Ghaziabad');
map.delete('city');
console.log(map.has('city'));
```

Example: To know the Map Size

```
const map = new Map();
map.set('first', 1);
map.set('second', 2);
console.log(map.size);
```

Example: Converting Map to Array

```
const map = new Map([['a', 1], ['b', 2], ['c', 3]]);
const arr = Array.from(map);
console.log(arr);
```

Example: Clearing All Entries in Map

```
const map = new Map([['name', 'Tushar Gupta'], ['age', 21]]);
map.clear();
console.log(map.size);
```

Example: Using Maps to Count Fruits

```
const arr = ['apple', 'banana', 'apple', 'orange', 'banana'];
const map = new Map();

arr.forEach(item => {
  map.set(item, (map.get(item) || 0) + 1);
});

console.log(map);
```

Storing Metadata for Objects: Since Map allows objects as keys, it's ideal for associating metadata with objects, without modifying the objects directly.

Example

```
let user = { name: 'Kavya Sharma' };
let metadata = new Map();
metadata.set(user, { role: 'student', learning: 'poor' });
console.log(metadata.get(user));
```

Complex Key Structures: If you need keys to be complex data types like objects or arrays, Map is the best choice.

Example

```
let coordinatesMap = new Map();
let coord = { x: 10, y: 20 };
coordinatesMap.set(coord, 'Device Location');
console.log(coordinatesMap)
```

Additional Use Cases

- **Caching Computation Results:** If you have expensive computations, you can cache the results using a Map, with the input as the key and the result as the value.
- **Tracking Visits in a Web Application:** Use a Map to store the number of times each user visits a page. The user ID can be the key, and the count can be the value.
- **Dictionaries with Non-String Keys:** You can create a map with complex objects or functions as keys, unlike an object which would convert non-string keys to strings.
- **Relational Mappings:** Use a Map to represent relationships between data. For example, you can map people to their friends, departments to employees, etc.

Note: Map in JavaScript is highly versatile and provides many advantages over objects, especially when working with key-value data that isn't restricted to string-based keys.

WeakMap: Object Key Store with Automatic Garbage Collection

- A WeakMap is similar to a Map in JavaScript, but with some important differences.
- It is a collection of key-value pairs where the keys must be objects and the values can be arbitrary values.
- The key distinction between a Map and a WeakMap is that in a WeakMap, the keys are held weakly, meaning that if there are no other references to the key object, it can be garbage collected.

Key Features of WeakMap

- **Weakly Held Keys:** The keys are objects, and if no other references to the key object exist, it will be garbage collected. This avoids memory leaks.
- **No Enumeration:** Unlike Map, you cannot iterate over the keys, values, or entries of a WeakMap. It is not iterable.
- **Only Object Keys:** Keys must be objects, not primitive data types.
- **No size Property:** WeakMap does not have a size property or methods to retrieve the number of elements.

Use Cases of WeakMap

The primary use case for WeakMap is associating metadata with objects while allowing those objects to be garbage collected if they are no longer referenced elsewhere in the application.

- **Private Data for Objects:** Use WeakMap to associate private data with objects without exposing the data.
- **Managing Resources:** WeakMaps are useful for managing resources where you don't want to worry about memory leaks. Once the object is no longer needed, the garbage collector cleans up the entries.
- **DOM Element Metadata:** WeakMaps can store metadata for DOM elements without preventing the elements from being garbage collected.
- **Caching Objects:** You can use WeakMap to cache data associated with objects and be sure that the cache will be cleared when the objects are no longer used.
- **Avoid Memory Leaks:** In situations where you're working with objects that need to be temporary, WeakMap prevents these objects from being kept alive unnecessarily by the map.

Basic Syntax

```
const weakMap = new WeakMap();

// Adding entries
        weakMap.set(keyObject, value);
// Accessing entries
        weakMap.get(keyObject);
// Checking existence
        weakMap.has(keyObject);
// Removing entries
        weakMap.delete(keyObject);
```

Example

```
const mymap = new WeakMap();
const obj = { name: 'Neha Rathore' };
mymap.set(obj, 'She is a good learner');
console.log(mymap.get(obj));
```

Example: Private Data Storage for Objects

```
const privateData = new WeakMap();
function User(name) {
        const obj = { secret: 'hidden data' };
        privateData.set(this, obj);   // stores the object (obj)as a private data reference for the current instance (this). The instance itself is used as the key.
          this.name = name;
}
User.prototype.getSecret = function() {
        return privateData.get(this).secret; // retrieves the private data object associated with the instance using privateData.get(this) and returns the secret property.
};
const user = new User('Neha Rathore');
console.log(user.getSecret());
console.log(user.name)
```

Example: Storing DOM Element Metadata

index.html

```
  <div style="width: 200px;height: 40px;background-color:green;text-align:center;">Click me!</div>
  <script src="index.js"></script>
```

index.js

```
const elementMetadata = new WeakMap();

const element = document.querySelector('div');
elementMetadata.set(element, { clicked: false });

element.addEventListener('click', () => {
        const metadata = elementMetadata.get(element);
        metadata.clicked = true;
        console.log('Element clicked:', metadata.clicked);
});
```

Example: Caching Expensive Computations

```
const cache = new WeakMap();

function expensiveComputation(obj) {
  if (cache.has(obj)) {
            return cache.get(obj);
  }
  const result = obj.value * 10; // Simulate computation
  cache.set(obj, result);
  return result;
}

const obj1 = { value: 5 };
console.log(expensiveComputation(obj1));
```

Example: Associating Metadata with DOM Nodes

```
const nodeMetadata = new WeakMap();

const div1 = document.createElement('div');
const div2 = document.createElement('div');

nodeMetadata.set(div1, { name: "first div" });
nodeMetadata.set(div2, { name: "second div" });

console.log(nodeMetadata.get(div1));
```

Example: Avoid Memory Leaks with Temporary Objects

```
let tempObject = { temp: 'data' };
const myMap = new WeakMap();

myMap.set(tempObject, 'temporary value');
console.log(myMap);
tempObject = null;
```

Example: Using WeakMap to Bind Data to UI Elements

```
const elementData = new WeakMap();

function setData(element, data) {
  elementData.set(element, data);
}

function getData(element) {
  return elementData.get(element);
}

const btn = document.createElement('button');
setData(btn, { action: 'submit' });
console.log(getData(btn));
```

Additional Use Cases

- **WeakMap as a Proxy for Private Class Properties:** It can be used as a way to simulate private variables in classes.
- **Tracking State for Multiple Objects:** Track states of various objects in large applications, especially when the object's lifecycle is short-lived.
- **Temporary Caches:** Use WeakMaps to store temporary cache entries that are automatically cleared when the associated object is no longer in use.
- **Storing Internal Implementation Details:** WeakMap is ideal for storing internal implementation details that should not be exposed outside an object.

Note: A WeakMap is a specialized collection designed for cases where keys need to be objects and should be garbage-collected when no longer referenced. It's great for managing metadata, temporary data storage, and preventing memory leaks by ensuring that objects are not retained longer than needed.