Set: A Collection of Unique Values

- The Set object is a new ES6 collection designed for holding unique values of any type (primitives or object references).
- It prevents duplicates automatically, making it highly useful for situations where the uniqueness of elements is required.
- The Set object in JavaScript lets you store unique values of any type, whether primitive values or object references.
- It's a great way to manage lists that require uniqueness, like tags, filters, or collections where duplication is unnecessary.

Why Use Set Over Arrays?

Arrays in JavaScript allow duplicates, and checking for duplicates is an expensive operation (indexOf or includes methods involve O(n) time complexity). Set, by contrast, automatically enforces uniqueness and provides faster membership checking.

- **Uniqueness Enforcement:** Values in a Set must be unique. If you add a value that already exists, the Set simply ignores the addition.
- **Performance:** Sets offer more efficient membership testing (O(1)) compared to arrays (O(n)).

Use Cases of Set

- **Removing duplicate values:** Set automatically eliminates duplicates from a list.
- **Tracking unique elements:** Useful for keeping a list of unique elements, such as visitors to a site.
- **Set operations:** You can perform mathematical set operations like union, intersection, and difference.
- **Filtering data:** Use Set to filter out certain values from a list, like excluding duplicates from an array.
- **Efficient membership testing:** Checking whether an element is part of a set is faster than checking in an array (O(1) vs. O(n)).
- **Keeping track of non-duplicate events:** Useful for handling events like user input, ensuring an action only happens once.
- **Tracking visited nodes in graph algorithms:** Prevents re-visiting nodes, helping to avoid infinite loops.

- **Implementing undo functionality:** Store states or actions without duplication.
- **Modeling many-to-many relationships:** Set ensures that each relationship is stored once without duplicates.
- **Unique random numbers:** To generate a collection of unique random numbers.

Basic Syntax

```
const mySet = new Set();

// Adding values to the Set
        mySet.add(value);
// Deleting a value from the Set
        mySet.delete(value);
// Checking if a value exists in the Set
        mySet.has(value);
// Getting the number of elements in the Set
        const size = mySet.size;
// Converting a Set to an Array
        const arrayFromSet = Array.from(mySet);
```

Example: Creating a Set

```
let mySet = new Set([1, 2, 3, 4, 5, 5]);
console.log(mySet);
```

Example: Adding elements to a Set

```
let mySet = new Set();
mySet.add("Neha");
mySet.add("Tushar");
mySet.add("Neha"); // Duplicate gets ignored
console.log(mySet);
```

Example: Removing duplicates from an array

```
let numbers = [1, 2, 2, 3, 4, 4, 5];
let uniqueNumbers = [...new Set(numbers)];
console.log(uniqueNumbers);
```

Example: Checking if an element exists in a Set

```
let mySet = new Set([1, 2, 3]);
console.log(mySet.has(2));
console.log(mySet.has(4));
```

Example: Removing an element from a Set

```
let mySet = new Set([1, 2, 3, 4]);
mySet.delete(3);
console.log(mySet);
```

Example: Clearing all elements in a Set

```
let mySet = new Set([1, 2, 3]);
mySet.clear();
console.log(mySet);
```

Example: Iterating over a Set

```
let studentSet = new Set(['Neha', 'Tushar', 'Sameer', 'Amit']);
for (let student of studentSet) {
   console.log(student);
}
```

Example: Converting Set to Array

```
let mySet = new Set([1, 2, 3, 4]);
let arr = Array.from(mySet);
console.log(arr);
```

Example: Union of two Sets

```
let setA = new Set([1, 2, 3]);
let setB = new Set([3, 4, 5]);
let union = new Set([...setA, ...setB]);
console.log(union);
```

Example: Intersection of two Sets

```
let setA = new Set([1, 2, 3]);
let setB = new Set([2, 3, 4]);
let intersection = new Set([...setA].filter(x => setB.has(x)));
console.log(intersection);
```

Example: Difference between two Sets

```
let setA= new Set([1, 2, 3, 4]);
let setB = new Set([3, 4, 5, 6]);
let difference = new Set([...setA].filter(x => !setB.has(x)));
console.log(difference);
```

Example: Symmetric Difference of two Sets

The symmetric difference is all elements that are in either of the two sets, but not in both.

```
let setA = new Set([1, 2, 3, 4]);
let setB = new Set([3, 4, 5, 6]);
let symmetricDifference = new Set([...setA].filter(x =>
!setB.has(x)).concat([...setB].filter(x => !setA.has(x))));
console.log(symmetricDifference);
```

Example: Using forEach() with Sets

```
let mySet = new Set(['a', 'b', 'c', 'd', 'e']);
mySet.forEach((value) => {
    console.log(value);
});
```

Example: Removing duplicates from a string

```
let string = "hello world";
let uniqueChars = [...new Set(string)].join('');
console.log(uniqueChars);
```

Example: Counting unique values in an array

```
let items = ['chair', 'table', 'chair', 'bed', 'almirah'];
let availableItems = new Set(items).size;
console.log(availableItems);
```

Example: Converting a Set to a string

```
let mySet = new Set([1, 2, 3, 4]);
let setString = [...mySet].join(', ');
console.log(setString);
```

Example: Using Sets to store objects

```
let obj1 = {name: 'Neha'};
let obj2 = {name: 'Tushar'};
let mySet = new Set([obj1, obj2]);
console.log(mySet.has(obj1));
console.log(mySet.has({name: 'Neha'}));
```

Example: Using Sets for simple caching

```
let itemCache = new Set();

function addToCache(item) {
   if (!itemCache.has(item)) {
      itemCache.add(item);
      console.log(`${item} added to cache`);
   } else {
      console.log(`${item} is already in the cache`);
   }
}

addToCache('chair');
addToCache('table');
addToCache('chair');
```

Example: Converting a Set of Sets to an Array of Arrays

```
let mySet = new Set([new Set([1, 2]), new Set([3, 4])]);
let arrayOfArrays = [...mySet].map(innerSet => [...innerSet]);
console.log(arrayOfArrays);
```

Example: Detecting duplicates with Sets

```
let array = [1, 2, 3, 4, 4, 5];
let hasDuplicates = new Set(array).size !== array.length;
console.log(hasDuplicates);
```

Note: The versatility of Set makes it useful in a wide range of applications, particularly when dealing with unique values or items. Whether for managing permissions, preventing duplicates in user inputs, or maintaining unique collections in applications, Set provides a simple and efficient way to handle such scenarios.

WeakSet: A Set of Objects with Weak References

- WeakSet is a variation of Set, where:
  - It can only hold objects, not primitive values.
  - The references to the objects are weak, meaning they don't prevent garbage collection.
- WeakSet is a special type of collection in JavaScript that allows you to store objects without preventing them from being garbage collected.
- Unlike Set, which holds strong references to its elements, WeakSet only holds weak references, meaning that if there are no other references to the object stored in the WeakSet, it can be garbage collected.

Why Use WeakSet?

The weak reference mechanism allows objects in a WeakSet to be garbage collected if there are no other references to them in the program. This is useful for managing object lifecycles, where you don't want to explicitly remove them from a collection when they're no longer needed.

WeakSet's Behavior

- **Non-Enumerable:** Like WeakMap, the contents of a WeakSet cannot be iterated over because the objects may disappear at any time due to garbage collection.
- **Efficient Object Tracking:** WeakSet is perfect for keeping track of objects without affecting their garbage collection, making it ideal for temporary or auxiliary data storage.

Use Cases of WeakSet

- **Memory Management:** Use WeakSet to store objects without preventing them from being garbage collected, helping manage memory usage efficiently.
- **Tracking Object States:** Store metadata or states of objects without affecting their lifecycle.
- **Event Listeners:** Use WeakSet to keep track of listeners without preventing them from being garbage collected when no longer needed.
- **Caching:** Cache data or objects temporarily without keeping them alive unnecessarily.
- **Unique Object References:** Track unique instances of objects, ensuring that the stored references do not interfere with garbage collection.

- **Building Private Data Structures:** Store private data associated with objects without exposing that data publicly.
- **Preventing Memory Leaks:** Use WeakSet to hold references to objects in cases where you want to ensure they can be collected when no longer used.
- **Implementation of Observer Patterns:** Track observers in a publish-subscribe model without preventing them from being garbage collected.
- **Temporary Object Relationships:** Use for temporary relationships between objects that should not affect garbage collection.
- **Managing DOM Elements:** Store references to DOM elements that may be removed, allowing them to be collected when they are no longer needed.

Example: Adding Objects to a WeakSet:

```
let obj = {};
let myweakset = new WeakSet();
myweakset.add(obj);
console.log(myweakset.has(obj));
```

Example: WeakSet with Garbage Collection

```
let obj = {};
let myweakset = new WeakSet();
weakset.add(obj);
obj = null; // Remove the reference
        // obj can now be garbage collected, and myweakset won't prevent it
```

Example: Checking for Existence

```
let obj = { name: 'Tushar' };
let myweakset = new WeakSet();
myweakset.add(obj);
console.log(myweakset.has(obj));
```

Example: Removing Objects from a WeakSet

```
let obj = { name: 'Neha' };
let myweakset = new WeakSet([obj]);
myweakset.delete(obj);
console.log(myweakset.has(obj));
```

Example: WeakSet and Event Listeners

index.html

```
<button id="myElement">Click Me!</button>
<button id="removeElement">Remove Button</button>
<p id="status"></p>
```

index.js

```
const listeners = new WeakSet();
// Define the event handler function
const eventHandler = (event) => {
    document.getElementById("status").textContent = "Button clicked!";
};
// Get the button element from the DOM
const element = document.getElementById('myElement');
// Add the event handler to the WeakSet and set it as a click event listener
listeners.add(eventHandler);
element.addEventListener('click', eventHandler);
// Remove the element from the DOM when "Remove Button" is clicked
const removeButton = document.getElementById('removeElement');
removeButton.addEventListener('click', () => {
    element.remove();
    document.getElementById("status").textContent = "Button removed!";
    // The eventHandler can now be garbage collected because the element is removed
});
```

Example: WeakSet for Private Data

```
<p id="output"></p>
  <script>
    // Create a WeakSet to store private data
    const privateData = new WeakSet();
    // Function to create a user and add the user object to the WeakSet
    function createUser(name) {
      const user = { name }; // Create user object
      privateData.add(user); // Add user to the WeakSet
      return user;
    }
```

```
// Create users
const user1 = createUser('Neha');
const user2 = createUser('Tushar');

// Check if privateData has the user
const outputElement = document.getElementById('output');
outputElement.textContent = `user1 available? ${privateData.has(user1)}\n`;
outputElement.textContent += `user2 available? ${privateData.has(user2)}\n`;

// After removing the reference to user1, it can be garbage collected
user1 = null;

// Trying to check again for user1 after making it null (but note WeakSet is not enumerable)
outputElement.textContent += "User1 has been set to null, eligible for garbage collection.";
</script>
```