Set Data Structure in JavaScript

A **Set** in JavaScript is a built-in data structure that stores a **collection of unique values** — meaning no duplicate elements are allowed. It can hold values of any type, whether primitive or object references.
You can store **any type of data**: numbers, strings, objects, booleans, and even other collections (like arrays or maps).

1. Creating a Set

1.1 Empty Set

```
const emptySet = new Set();
console.log(emptySet); // Set(0) {}
```

- Creates a blank Set.
- Size = 0 initially.

1.2 From an Array

```
const numbers = new Set([1,2,3,3,4]);
console.log(numbers); // Set(4) {1,2,3,4}
```

- Duplicates in the array are automatically removed.
- Preserves **insertion order**.

1.3 From a String

```
const strSet = new Set("hello");
console.log(strSet); // Set(4) {'h','e','l','o'}
```

- Only unique letters are kept.

1.4 From Any Iterable

```
const map = new Map([[1,'a'],[2,'b']]);
const setFromMap = new Set(map.keys());
console.log(setFromMap); // Set(2) {1,2}
```

• Can create a Set from **Map keys, Map values, arrays, or any iterable**.

2. Properties of Set

2.1 size

• Returns the number of unique elements.

```
const s = new Set([1,2,3]);
console.log(s.size); // 3
```

3. Methods of Set

3.1 add(value)

• Adds a value to the Set.

• Returns the Set itself, allowing **chaining**.

```
const s = new Set();
s.add(1).add(2).add(3);
console.log(s); // Set(3) {1,2,3}
```

3.2 delete(value)

• Removes a specific value.

• Returns `true` if value existed and removed, `false` otherwise.

```
s.delete(2);
console.log(s); // Set(2) {1,3}
```

3.3 has(value)

• Checks if value exists in the Set.

```
console.log(s.has(1)); // true
console.log(s.has(5)); // false
```

3.4 clear()

• Removes all elements from the Set.

```
s.clear();
console.log(s); // Set(0) {}
```

4. Iterating Over a Set

4.1 for...of

```
const s = new Set([1,2,3]);
for (let value of s) {
  console.log(value);
}
// Output: 1 2 3
```

4.2 forEach()

```
s.forEach(value => console.log(value));
```

4.3 keys(), values(), entries()

- `keys()` – returns iterator for values (because Set has no keys, same as values).
- `values()` – returns iterator for values.
- `entries()` – returns `[value, value]` pairs (for compatibility with Map).

```
for (let k of s.keys()) console.log(k);
for (let v of s.values()) console.log(v);
for (let [a,b] of s.entries()) console.log(a,b);
// Output: 1 1, 2 2, 3 3
```

5. Converting Sets

5.1 Set → Array

```
const arr = [...s];
console.log(arr); // [1,2,3]
```

- Spread operator `...` or `Array.from()` can convert Set to Array.

5.2 Array → Set

```
const arr2 = [1,2,2,3,3];
const uniqueSet = new Set(arr2);
console.log([...uniqueSet]); // [1,2,3]
```

- Automatically removes duplicates.

6. Set Operations (Math Style)

6.1 Union

```
const a = new Set([1,2,3]);
const b = new Set([3,4,5]);
const union = new Set([...a, ...b]);
console.log(union); // Set(5) {1,2,3,4,5}
```

6.2 Intersection

```
const intersection = new Set([...a].filter(x => b.has(x)));
console.log(intersection); // Set(1) {3}
```

6.3 Difference

```
const difference = new Set([...a].filter(x => !b.has(x)));
console.log(difference); // Set(2) {1,2}
```

6.4 Symmetric Difference

```
const symDiff = new Set([...a, ...b].filter(x => !(a.has(x) && b.has(x))));
console.log(symDiff); // Set(4) {1,2,4,5}
```

7. Advanced Executions

7.1 Chaining

```
const s1 = new Set([1,2,3]);
s1.add(4).delete(2);
console.log(s1); // Set(3) {1,3,4}
```

7.2 Remove duplicates from string

```
const str = "banana";
const uniqueLetters = [...new Set(str)].join('');
console.log(uniqueLetters); // "ban"
```

7.3 Remove duplicates from object array

```
const objArr = [{id:1},{id:2},{id:1}];
const uniqueIds = [...new Set(objArr.map(o => o.id))];
console.log(uniqueIds); // [1,2]
```

8. Iterables and Array Methods

• You can **filter, map, reduce** Sets by converting them to arrays:

```
const s = new Set([1,2,3,4]);
const filtered = [...s].filter(x => x>2);
console.log(filtered); // [3,4]
```

• Spread operator can be used for function calls:

```
function sum(a,b,c){ return a+b+c; }
const nums = new Set([1,2,3]);
console.log(sum(...nums)); // 6
```

9. Real-World Use Cases

1. Remove duplicates from arrays or strings.
2. Fast lookup (`has()`) instead of array search.
3. Maintain **unique IDs or keys**.
4. Implement **union, intersection, difference** operations.
5. Keep **unique event listeners**.

10. Key Differences vs Array

Feature	Set	Array
Duplicates	❌ Not allowed	✅ Allowed
Access by index	❌ No	✅ Yes
Order	✅ Maintained	✅ Maintained
Search	⚡ Fast (`has()`)	🐢 Slow (`indexOf`)