**Part 1 — JavaScript class (baby steps)**

Think of a class as a blueprint for making many similar objects (instances).  
Picture: blueprint → you use it to build many houses. Each house has its own rooms (properties) and doors (methods).

**class Dog {**

**constructor(name) { // runs when you create a Dog**

**this.name = name; // store this dog's name**

**}**

**bark() { // a method every Dog has**

**console.log(this.name + " says woof");**

**}**

**}**

**let rufus = new Dog("Rufus"); // make an instance**

**rufus.bark(); // Rufus says woof**

**Part 2 — Build MyHashSet (step-by-step)**

**Goal (very simple)**

Support these operations:

* add(key) — put key into set
* remove(key) — remove key
* contains(key) — check if key exists

We **won’t use** built-in Set or Map. We’ll use a simple hash + buckets.

**Idea in one sentence**

Use an array of buckets; for a key compute index = hash(key), store the key in buckets[index] (if not already there). To check or remove, look only in that bucket.

Implementation (clear code)

class MyHashSet {

//default val for bucketCount

constructor(bucketCount = 1000) {

this.bucketCount = bucketCount;

// create bucketCount empty buckets (arrays)

this.buckets = Array.from({ length: bucketCount }, () => []);

}

// simple hash function (keeps index >= 0)

\_hash(key) {

return Math.abs(key) % this.bucketCount;

}

add(key) {

const idx = this.\_hash(key);

const bucket = this.buckets[idx];

if (!bucket.includes(key)) { // only add if not already present

bucket.push(key);

}

}

remove(key) {

const idx = this.\_hash(key);

const bucket = this.buckets[idx];

const pos = bucket.indexOf(key);

if (pos !== -1) {

bucket.splice(pos, 1); // remove the key

}

}

contains(key) {

const idx = this.\_hash(key);

return this.buckets[idx].includes(key);

}

}

**How to run a simple test:**

const mySet = new MyHashSet();

mySet.add(1);

mySet.add(2);

console.log(mySet.contains(1)); // true

console.log(mySet.contains(3)); // false

mySet.add(2);

console.log(mySet.contains(2)); // true

mySet.remove(2);

console.log(mySet.contains(2)); // false

**Quick checklist — why each line in the class exists**

* constructor(...) — sets up our buckets (like building empty boxes)
* \_hash(key) — computes which box to use
* add(key) — check bucket, push if not present
* remove(key) — find and remove from bucket
* contains(key) — check the bucket

**🧩 Step 1: Why do we need a “hash”?**

When we create our own **HashSet**, we need a way to decide **where** (in which “bucket”) to store a given number or key.

For example:

* Suppose our HashSet has **10 buckets** (like 10 boxes).
* When someone adds a key like 15, we can’t just drop it anywhere.
* We need to decide which box (0–9) to put it in.

So we use a **hash function** — a formula that turns a value into an index (a number between 0 and 9).

Example simple hash formula:

key % 10

→ If key = 15 → 15 % 10 = 5 → we store it in bucket number 5.

This is why we have something like:

\_hash(key) {

return key % this.size;

}

It’s just a **function to calculate the bucket position**.

**🧠 Step 2: Why the underscore before \_hash?**

In JavaScript, when we name something with an underscore \_ (like \_hash),  
it’s a **common naming convention** meaning:

“Hey! This is a **private** helper method, not meant to be used directly outside the class.”

So \_hash() means it’s an **internal helper** — used *inside the class only* to compute the hash.

You could name it without underscore (hash()), and it would still work,  
but developers use \_ to make it clear that it's not part of the public API.

**🧱 Step 3: Visualizing how it’s used**

Imagine you have a HashSet with 10 buckets:

this.buckets = [ [], [], [], [], [], [], [], [], [], [] ]

When you call:

add(15)

👉 It calls \_hash(15)  
👉 \_hash(15) returns 5  
👉 It adds 15 into buckets[5]

Now when you check:

contains(15)

👉 It again calls \_hash(15) → gets index 5  
👉 Then checks if 15 is inside buckets[5].

**🧩 Step 1: What \_ actually *means* in JavaScript**

In JavaScript, the underscore (\_) **does NOT** make something *truly private*.  
It’s **just a naming convention** — a signal to other developers that:

“This is meant to be used *inside the class only*, not by the outside world.”

So if you write this:

class MyHashSet {

\_addHelper() {

console.log("I'm a helper");

}

}

You **can still access it** from an object like this:

const obj = new MyHashSet();

obj.\_addHelper(); // ✅ Works fine

👉 It’s **not truly private**, it’s just *a way of saying* “please don’t touch this method from outside.”

**🧠 Step 2: So what’s the difference between \_hash and add?**

| **Method Name** | **Used For** | **Can be called outside?** | **Meaning** |
| --- | --- | --- | --- |
| \_hash | Internal helper | ✅ Yes (but shouldn’t) | Developer convention (helper) |
| add | Main public function | ✅ Yes | Public method (meant to be called by user) |

So in our MyHashSet example:

const set = new MyHashSet();

set.add(10); // ✅ Intended for user

set.\_hash(10); // ⚠️ Works, but not recommended (helper)

| **Name style** | **Can call outside** | **Meaning** |
| --- | --- | --- |
| \_hash() | ✅ Yes | “Please don’t use me directly” (helper) |
| #hash() | ❌ No | Truly private, only for internal use |
| add() | ✅ Yes | Public API method |