Map

Baseline Widely available *

The `Map` object holds key-value pairs and remembers the original insertion order of the keys. Any value (both objects and primitive values) may be used as either a key or a value.

In this article

Try it

JavaScript Demo: Map

```
1 const map = new Map();
2
3 map.set("a", 1);
```

```
map.set("b", 2);
map.set("c", 3);

console.log(map.get("a"));
// Expected output: 1

map.set("a", 97);

console.log(map.get("a"));
// Expected output: 97

console.log(map.size);
// Expected output: 3

map.delete("b");

console.log(map.size);
// Expected output: 2

```

Run Reset

Description

`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` object is iterated by key-value pairs — a `for...of` loop returns a 2-member array of `[key, value]` for each iteration. Iteration happens in *insertion order*, which corresponds to the order in which each key-value pair was first inserted into the map by the `set()` method (that is, there wasn't a key with the same value already in the map when `set()` was called).

The specification requires maps to be implemented "that, on average, provide access times that are sublinear on the number of elements in the collection". Therefore, it could be

represented internally as a hash table (with O(1) lookup), a search tree (with O(log(N)) lookup), or any other data structure, as long as the complexity is better than O(N).

Key equality

Value equality is based on the SameValueZero algorithm. (It used to use SameValue, which treated `0` and `-0` as different. Check browser compatibility.) This means `NaN` is considered the same as `NaN` (even though `NaN !== NaN`) and all other values are considered equal according to the semantics of the `===` operator.

Objects vs. Maps

`Object` is similar to `Map`—both let you set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. For this reason (and because there were no built-in alternatives), `Object` has been used as `Map` historically.

However, there are important differences that make `Map` preferable in some cases:

	Map	Object
Accidental Keys	A `Map` does not contain any keys by default. It only contains what is explicitly put into it.	An `Object` has a prototype, so it contains default keys that could collide with your own keys if you're not careful. **Note:** This can be bypassed by using `Object.create(null)`, but this is seldom done.

	Map	Object
Security	A `Map` is safe to use with user-provided keys and values.	Setting user-provided key-value pairs on an `Object` may allow an attacker to override the object's prototype, which can lead to object injection attacks. Like the accidental keys issue, this can also be mitigated by using a `null`-prototype object.
Key Types	A `Map`'s keys can be any value (including functions, objects, or any primitive).	The keys of an `Object` must be either a `String` or a `Symbol`.

	Map	Object
Key Order	The keys in `Map` are ordered in a straightforward way: A `Map` object iterates entries, keys, and values in the order of entry insertion.	Although the keys of an ordinary `Object` are ordered now, this was not always the case, and the order is complex. As a result, it's best not to rely on property order. The order was first defined for own properties only in ECMAScript 2015; ECMAScript 2020 defines order for inherited properties as well. But note that no single mechanism iterates **all** of an object's properties; the various mechanisms each include different subsets of properties. (`for-in` includes only enumerable string-keyed properties; `Object.keys` includes only own, enumerable, string-keyed properties; `Object.getOwnPropertyNames` includes own, string-keyed properties even if non-enumerable; `Object.getOwnPropertySymbols` does the same for just `Symbol`-keyed properties, etc.)
Size	The number of items in a `Map` is easily retrieved from its `size` property.	Determining the number of items in an `Object` is more roundabout and less efficient. A common way to do it is through the `length` of the array returned from `Object.keys()`.
Iteration	A `Map` is an iterable, so it can be directly iterated.	`Object` does not implement an iteration protocol, and so objects are not directly iterable using the JavaScript for...of statement (by default).

	Map	Object
		Note: • An object can implement the iteration protocol, or you can get an iterable for an object using `Object.keys` or `Object.entries`. • The for...in statement allows you to iterate over the *enumerable* properties of an object.
Performance	Performs better in scenarios involving frequent additions and removals of key-value pairs.	Not optimized for frequent additions and removals of key-value pairs.
Serialization and parsing	No native support for serialization or parsing. (But you can build your own serialization and parsing support for `Map` by using `JSON.stringify()` with its *replacer* argument, and by using `JSON.parse()` with its *reviver* argument. See the Stack Overflow	Native support for serialization from `Object` to JSON, using `JSON.stringify()`. Native support for parsing from JSON to `Object`, using `JSON.parse()`.

	Map	Object
	question How do you JSON.stringify an ES6 Map?).	

Setting object properties

Setting Object properties works for Map objects as well, and can cause considerable confusion.

Therefore, this appears to work in a way:

JS Copy

```
const wrongMap = new Map();
wrongMap["bla"] = "blaa";
wrongMap["bla2"] = "blaaa2";

console.log(wrongMap); // Map { bla: 'blaa', bla2: 'blaaa2' }
```

But that way of setting a property does not interact with the Map data structure. It uses the feature of the generic object. The value of 'bla' is not stored in the Map for queries. Other operations on the data fail:

JS Copy

```
wrongMap.has("bla"); // false
wrongMap.delete("bla"); // false
console.log(wrongMap); // Map { bla: 'blaa', bla2: 'blaaa2' }
```

The correct usage for storing data in the Map is through the `set(key, value)` method.

JS Copy

```
const contacts = new Map();
contacts.set("Jessie", { phone: "213-555-1234", address: "123 N 1st Ave" });
contacts.has("Jessie"); // true
contacts.get("Hilary"); // undefined
contacts.set("Hilary", { phone: "617-555-4321", address: "321 S 2nd St" });
contacts.get("Jessie"); // {phone: "213-555-1234", address: "123 N 1st Ave"}
contacts.delete("Raymond"); // false
contacts.delete("Jessie"); // true
console.log(contacts.size); // 1
```

Map-like browser APIs

Browser `Map`-like objects (or "maplike objects") are Web API interfaces that behave in many ways like a `Map`.

Just like `Map`, entries can be iterated in the same order that they were added to the object. `Map`-like objects and `Map` also have properties and methods that share the same name and behavior. However unlike `Map` they only allow specific predefined types for the keys and values of each entry.

The allowed types are set in the specification IDL definition. For example, `RTCStatsReport` is a `Map`-like object that must use strings for keys and objects for values. This is defined in the specification IDL below:

WEBIDL Copy

```
interface RTCStatsReport {
  readonly maplike<DOMString, object>;
};
```

`Map`-like objects are either read-only or read-writable (see the `readonly` keyword in the IDL above).

- Read-only `Map`-like objects have the property `size`, and the methods: `entries()`, `forEach()`, `get()`, `has()`, `keys()`, `values()`, and `Symbol.iterator()`.
- Writeable `Map`-like objects additionally have the methods: `clear()`, `delete()`, and `set()`.

The methods and properties have the same behavior as the equivalent entities in `Map`, except for the restriction on the types of the keys and values.

The following are examples of read-only `Map`-like browser objects:

- `AudioParamMap`
- `RTCStatsReport`
- `EventCounts`
- `KeyboardLayoutMap`
- `MIDIInputMap`
- `MIDIOutputMap`

Constructor

`Map()`

Creates a new `Map` object.

Static properties

`Map[Symbol.species]`

The constructor function that is used to create derived objects.

Static methods

`Map.groupBy()`

Groups the elements of a given iterable using the values returned by a provided callback function. The final returned `Map` uses the unique values from the test function as keys, which can be used to get the array of elements in each group.

Instance properties

These properties are defined on `Map.prototype` and shared by all `Map` instances.

`Map.prototype.constructor`

The constructor function that created the instance object. For `Map` instances, the initial value is the `Map` constructor.

`Map.prototype.size`

Returns the number of key/value pairs in the `Map` object.

`Map.prototype[Symbol.toStringTag]`

The initial value of the `[Symbol.toStringTag]` property is the string `"Map"`. This property is used in `Object.prototype.toString()`.

Instance methods

`Map.prototype.clear()`

Removes all key-value pairs from the `Map` object.

`Map.prototype.delete()`

Returns `true` if an element in the `Map` object existed and has been removed, or `false` if the element does not exist. `map.has(key)` will return `false` afterwards.

`Map.prototype.entries()`

Returns a new Iterator object that contains a two-member array of `[key, value]` for each element in the `Map` object in insertion order.

`Map.prototype.forEach()`

Calls `callbackFn` once for each key-value pair present in the `Map` object, in insertion order. If a `thisArg` parameter is provided to `forEach`, it will be used as the `this` value for each callback.

`Map.prototype.get()`

Returns the value associated to the passed key, or `undefined` if there is none.

`Map.prototype.has()`

Returns a boolean indicating whether a value has been associated with the passed key in the `Map` object or not.

`Map.prototype.keys()`

Returns a new Iterator object that contains the keys for each element in the `Map` object in insertion order.

`Map.prototype.set()`

Sets the value for the passed key in the `Map` object. Returns the `Map` object.

`Map.prototype.values()`

Returns a new Iterator object that contains the values for each element in the `Map` object in insertion order.

`Map.prototype[Symbol.iterator]()`

Returns a new Iterator object that contains a two-member array of `[key, value]` for each element in the `Map` object in insertion order.

Examples

Using the Map object

JS Copy

```
const myMap = new Map();

const keyString = "a string";
const keyObj = {};
const keyFunc = () => {};

// setting the values
myMap.set(keyString, "value associated with 'a string'");
myMap.set(keyObj, "value associated with keyObj");
myMap.set(keyFunc, "value associated with keyFunc");

console.log(myMap.size); // 3

// getting the values
console.log(myMap.get(keyString)); // "value associated with 'a string'"
console.log(myMap.get(keyObj)); // "value associated with keyObj"
console.log(myMap.get(keyFunc)); // "value associated with keyFunc"

console.log(myMap.get("a string")); // "value associated with 'a string'",
because keyString === 'a string'
console.log(myMap.get({})); // undefined, because keyObj !== {}
console.log(myMap.get(() => {})); // undefined, because keyFunc !== () =>
{}
```

Using NaN as Map keys

`NaN` can also be used as a key. Even though every `NaN` is not equal to itself (`NaN !== NaN` is true), the following example works because `NaN`s are indistinguishable from each other:

JS Copy

```
const myMap = new Map();
myMap.set(NaN, "not a number");
```

```
myMap.get(NaN);
// "not a number"

const otherNaN = Number("foo");
myMap.get(otherNaN);
// "not a number"
```

Iterating Map with for...of

Maps can be iterated using a `for...of` loop:

JS Copy

```
const myMap = new Map();
myMap.set(0, "zero");
myMap.set(1, "one");

for (const [key, value] of myMap) {
  console.log(`${key} = ${value}`);
}
// 0 = zero
// 1 = one

for (const key of myMap.keys()) {
  console.log(key);
}
// 0
// 1

for (const value of myMap.values()) {
  console.log(value);
}
// zero
// one
```

```
for (const [key, value] of myMap.entries()) {
  console.log(`${key} = ${value}`);
}
// 0 = zero
// 1 = one
```

Iterating Map with forEach()

Maps can be iterated using the `forEach()` method:

JS Copy

```
myMap.forEach((value, key) => {
  console.log(`${key} = ${value}`);
});
// 0 = zero
// 1 = one
```

Relation with Array objects

JS Copy

```
const kvArray = [
  ["key1", "value1"],
  ["key2", "value2"],
];

// Use the regular Map constructor to transform a 2D key-value Array into
a map
const myMap = new Map(kvArray);

console.log(myMap.get("key1")); // "value1"
```

```
// Use Array.from() to transform a map into a 2D key-value Array
console.log(Array.from(myMap)); // Will show you exactly the same Array as
kvArray

// A succinct way to do the same, using the spread syntax
console.log([...myMap]);

// Or use the keys() or values() iterators, and convert them to an array
console.log(Array.from(myMap.keys())); // ["key1", "key2"]
```

Cloning and merging Maps

Just like `Array`s, `Map`s can be cloned:

JS Copy

```
const original = new Map([[1, "one"]]);

const clone = new Map(original);

console.log(clone.get(1)); // one
console.log(original === clone); // false (useful for shallow comparison)
```

> **Note:** Keep in mind that *the data itself* is not cloned. In other words, it is only a shallow copy of the `Map`.

Maps can be merged, maintaining key uniqueness:

JS Copy

```
const first = new Map([
  [1, "one"],
  [2, "two"],
```

```
  [3, "three"],
]);

const second = new Map([
  [1, "uno"],
  [2, "dos"],
]);

// Merge two maps. The last repeated key wins.
// Spread syntax essentially converts a Map to an Array
const merged = new Map([...first, ...second]);

console.log(merged.get(1)); // uno
console.log(merged.get(2)); // dos
console.log(merged.get(3)); // three
```

Maps can be merged with Arrays, too:

JS Copy

```
const first = new Map([
  [1, "one"],
  [2, "two"],
  [3, "three"],
]);

const second = new Map([
  [1, "uno"],
  [2, "dos"],
]);

// Merge maps with an array. The last repeated key wins.
const merged = new Map([...first, ...second, [1, "un"]]);
```

```
console.log(merged.get(1)); // un
console.log(merged.get(2)); // dos
console.log(merged.get(3)); // three
```

Specifications

Specification
ECMAScript® 2026 Language Specification # sec-map-objects

Browser compatibility

Report problems with this compatibility data • View data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS	Deno	Node.js
`Map`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
[Symbol.iterator]	✓ 38	✓ 12	✓ 36 …	✓ 25	✓ 10	✓ 38	✓ 36 …	✓ 25	✓ 10	✓ 3	✓ 38	✓ 10	✓ 1	✓ 0.12
[Symbol.species]	✓ 51	✓ 13	✓ 41	✓ 38	✓ 10	✓ 51	✓ 41	✓ 41	✓ 10	✓ 5	✓ 51	✓ 10	✓ 1	✓ 6.5
`Map()` constructor	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`new Map(iterable)`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 9	✓ 38	✓ 14	✓ 25	✓ 9	✓ 3	✓ 38	✓ 9	✓ 1	✓ 0.12
`new Map(null)`	✓ 38	✓ 12	✓ 37	✓ 25	✓ 9	✓ 38	✓ 37	✓ 25	✓ 9	✓ 3	✓ 38	✓ 9	✓ 1	✓ 0.12
`clear`	✓ 38	✓ 12	✓ 19	✓ 25	✓ 8	✓ 38	✓ 19	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12

`delete`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`entries`	✓ 38	✓ 12	✓ 20	✓ 25	✓ 8	✓ 38	✓ 20	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`forEach`	✓ 38	✓ 12	✓ 25	✓ 25	✓ 8	✓ 38	✓ 25	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`get`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`groupBy`	✓ 117	✓ 117	✓ 119	✓ 103	✓ 17.4 •••	✓ 117	✓ 119	✓ 78	✓ 17.4 •••	✓ 24	✓ 117	✓ 17.4 •••	✓ 1.37	✓ 21
`has`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
Key equality for `-0` and `0`	✓ 38	✓ 12	✓ 29	✓ 25	✓ 9	✓ 38	✓ 29	✓ 25	✓ 9	✓ 3	✓ 38	✓ 9	✓ 1	✓ 4
`keys`	✓ 38	✓ 12	✓ 20	✓ 25	✓ 8	✓ 38	✓ 20	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`set`	✓ 38	✓ 12	✓ 13	✓ 25	✓ 8	✓ 38	✓ 14	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`size`	✓ 38	✓ 12	✓ 19 *	✓ 25	✓ 8	✓ 38	✓ 19 *	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12
`values`	✓ 38	✓ 12	✓ 20	✓ 25	✓ 8	✓ 38	✓ 20	✓ 25	✓ 8	✓ 3	✓ 38	✓ 8	✓ 1	✓ 0.12

Tip: you can click/tap on a cell for more information.

✓ Full support * See implementation notes. Uses a non-standard name.

••• Has more compatibility info.

See also

- Polyfill for `Map` in `core-js`
- es-shims polyfill of `Map`
- `Set`

- `WeakMap`
- `WeakSet`

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute

This page was last modified on Jul 20, 2025 by MDN contributors.

View this page on GitHub • Report a problem with this content

mdn_

Your blueprint for a better internet.

MDN

About

Blog

Mozilla careers

Advertise with us

MDN Plus

Contribute

MDN Community

Community resources

Writing guidelines

MDN Discord

MDN on GitHub

Developers

Web technologies

Learn web development

Guides

Tutorials

Glossary

Product help Hacks blog

Mozilla Website Privacy Notice Cookies Legal Community Participation Guidelines