mdn web docs

Memory management

Low-level languages like C, have manual memory management primitives such as `malloc()` and `free()`. In contrast, JavaScript automatically allocates memory when objects are created and frees it when they are not used anymore (*garbage collection*). This automaticity is a potential source of confusion: it can give developers the false impression that they don't need to worry about memory management.

Memory life cycle

Regardless of the programming language, the memory life cycle is pretty much always the same:

1. Allocate the memory you need
2. Use the allocated memory (read, write)
3. Release the allocated memory when it is not needed anymore

The second part is explicit in all languages. The first and last parts are explicit in low-level languages but are mostly implicit in high-level languages like JavaScript.

Allocation in JavaScript

Value initialization

In order to not bother the programmer with allocations, JavaScript will automatically allocate memory when values are initially declared.

JS

```js
const n = 123; // allocates memory for a number
const s = "azerty"; // allocates memory for a string

const o = {
  a: 1,
  b: null,
}; // allocates memory for an object and contained values

// (like object) allocates memory for the array and
// contained values
const a = [1, null, "abra"];

function f(a) {
  return a + 2;
} // allocates a function (which is a callable object)

// function expressions also allocate an object
someElement.addEventListener(
  "click",
  () => {
    someElement.style.backgroundColor = "blue";
  },
  false,
);
```

Allocation via function calls

Some function calls result in object allocation.

JS

```js
const d = new Date(); // allocates a Date object

const e = document.createElement("div"); // allocates a DOM element
```

Some methods allocate new values or objects:

JS

```js
const s = "azerty";
const s2 = s.substr(0, 3); // s2 is a new string
// Since strings are immutable values,
// JavaScript may decide to not allocate memory,
// but just store the [0, 3] range.

const a = ["ouais ouais", "nan nan"];
const a2 = ["generation", "nan nan"];
const a3 = a.concat(a2);
// new array with 4 elements being
// the concatenation of a and a2 elements.
```

Using values

Using values basically means reading and writing in allocated memory. This can be done by reading or writing the value of a variable or an object property or even passing an argument to a function.

Release when the memory is not needed anymore

The majority of memory management issues occur at this phase. The most difficult aspect of this stage is determining when the allocated memory is no longer needed.

Low-level languages require the developer to manually determine at which point in the program the allocated memory is no longer needed and to release it.

Some high-level languages, such as JavaScript, utilize a form of automatic memory management known as garbage collection (GC). The purpose of a garbage collector is to monitor memory allocation and determine when a block of allocated memory is no longer needed and reclaim it. This automatic process is an approximation since the general problem of determining whether or not a specific piece of memory is still needed is undecidable.

Garbage collection

As stated above, the general problem of automatically finding whether some memory "is not needed anymore" is undecidable. As a consequence, garbage collectors implement a restriction of a solution to the general problem. This section will explain the concepts that are necessary for understanding the main garbage collection algorithms and their respective limitations.

References

The main concept that garbage collection algorithms rely on is the concept of *reference*. Within the context of memory management, an object is said to reference another object if the former has access to the latter (either implicitly or explicitly). For instance, a JavaScript object has a reference to its prototype (implicit reference) and to its properties values (explicit reference).

In this context, the notion of an "object" is extended to something broader than regular JavaScript objects and also contain function scopes (or the global lexical scope).

Reference-counting garbage collection

> **Note:** no modern JavaScript engine uses reference-counting for garbage collection anymore.

This is the most naïve garbage collection algorithm. This algorithm reduces the problem from determining whether or not an object is still needed to determining if an object still has any other objects referencing it. An object is said to be "garbage", or collectible if there are zero references pointing to it.

For example:

JS

```
let x = {
  a: {
    b: 2,
  },
};
// 2 objects are created. One is referenced by the other as one of its properties.
// The other is referenced by virtue of being assigned to the 'x' variable.
// Obviously, none can be garbage-collected.

let y = x;
// The 'y' variable is the second thing that has a reference to the object.

x = 1;
// Now, the object that was originally in 'x' has a unique reference
// embodied by the 'y' variable.

let z = y.a;
// Reference to 'a' property of the object.
// This object now has 2 references: one as a property,
// the other as the 'z' variable.

y = "mozilla";
// The object that was originally in 'x' has now zero
// references to it. It can be garbage-collected.
// However its 'a' property is still referenced by
// the 'z' variable, so it cannot be freed.

z = null;
// The 'a' property of the object originally in x
// has zero references to it. It can be garbage collected.
```

There is a limitation when it comes to circular references. In the following example, two objects are created with properties that reference one another, thus creating a cycle. They will go out of scope after the function call has completed. At that point they become unneeded and their allocated memory should be reclaimed. However, the reference-counting algorithm will not consider them reclaimable since each of the two objects has at least one reference pointing to them, resulting in neither of them being marked for garbage collection. Circular references are a common cause of memory leaks.

JS

```js
function f() {
  const x = {};
  const y = {};
  x.a = y; // x references y
  y.a = x; // y references x

  return "azerty";
}

f();
```

Mark-and-sweep algorithm

This algorithm reduces the definition of "an object is no longer needed" to "an object is unreachable".

This algorithm assumes the knowledge of a set of objects called *roots*. In JavaScript, the root is the global object. Periodically, the garbage collector will start from these roots, find all objects that are referenced from these roots, then all objects referenced from these, etc. Starting from the roots, the garbage collector will thus find all *reachable* objects and collect all non-reachable objects.

This algorithm is an improvement over the previous one since an object having zero references is effectively unreachable. The opposite does not hold true as we have seen with circular references.

Currently, all modern engines ship a mark-and-sweep garbage collector. All improvements made in the field of JavaScript garbage collection (generational/incremental/concurrent/parallel garbage collection) over the last few years are implementation improvements of this algorithm, but not improvements over the garbage collection algorithm itself nor its reduction of the definition of when "an object is no longer needed".

The immediate benefit of this approach is that cycles are no longer a problem. In the first example above, after the function call returns, the two objects are no longer referenced by any resource that is reachable from the global object. Consequently, they will be found unreachable by the garbage collector and have their allocated memory reclaimed.

However, the inability to manually control garbage collection remains. There are times when it would be convenient to manually decide when and what memory is released. In order to release the memory of an object, it needs to be made explicitly unreachable. It is also not possible to programmatically trigger garbage collection in JavaScript — and will likely never be within the core language, although engines may expose APIs behind opt-in flags.

Configuring an engine's memory model

JavaScript engines typically offer flags that expose the memory model. For example, Node.js offers additional options and tools that expose the underlying V8 mechanisms for configuring and debugging memory issues. This configuration may not be available in browsers, and even less so for web pages (via HTTP headers, etc.).

The max amount of available heap memory can be increased with a flag:

BASH

```bash
node --max-old-space-size=6000 index.js
```

We can also expose the garbage collector for debugging memory issues using a flag and the Chrome Debugger:

BASH

```bash
node --expose-gc --inspect index.js
```

Data structures aiding memory management

Although JavaScript does not directly expose the garbage collector API, the language offers several data structures that indirectly observe garbage collection and can be used to manage memory usage.

WeakMaps and WeakSets

`WeakMap` and `WeakSet` are data structures whose APIs closely mirror their non-weak counterparts: `Map` and `Set`. `WeakMap` allows you to maintain a collection of key-value pairs, while `WeakSet` allows you to maintain a collection of unique values, both with performant addition, deletion, and querying.

`WeakMap` and `WeakSet` got the name from the concept of *weakly held* values. If `x` is weakly held by `y`, it means that although you can access the value of `x` via `y`, the mark-and-sweep algorithm won't consider `x` as reachable if nothing else *strongly holds* to it. Most data structures, except the ones discussed here, strongly holds to the objects passed in so that you can retrieve them at any time. The keys of `WeakMap` and `WeakSet` can be garbage-collected (for `WeakMap` objects, the values would then be eligible for garbage collection as well) as long as nothing else in the program is referencing the key. This is ensured by two characteristics:

- `WeakMap` and `WeakSet` can only store objects or symbols. This is because only objects are garbage collected — primitive values can always be forged (that is, `1 === 1` but `{} !== {}`), making them stay in the collection forever. Registered symbols (like `Symbol.for("key")`) can also be forged and thus not garbage collectable, but symbols created with `Symbol("key")` are garbage collectable. Well-known symbols like `Symbol.iterator` come in a fixed set and are unique throughout the lifetime of the program, similar to intrinsic objects such as `Array.prototype`, so they are also allowed as keys.

- `WeakMap` and `WeakSet` are not iterable. This prevents you from using `Array.from(map.keys()).length` to observe the liveliness of objects, or get hold of an arbitrary key which should otherwise be eligible for garbage collection. (Garbage collection should be as invisible as possible.)

In typical explanations of `WeakMap` and `WeakSet` (such as the one above), it's often implied that the key is garbage-collected first, freeing the value for garbage collection as well. However, consider the case of the value referencing the key:

JS

```js
const wm = new WeakMap();
const key = {};
wm.set(key, { key });
// Now `key` cannot be garbage collected,
// because the value holds a reference to the key,
// and the value is strongly held in the map!
```

If `key` is stored as an actual reference, it would create a cyclic reference and make both the key and value ineligible for garbage collection, even when nothing else references `key` — because if `key` is garbage collected, it means that at some particular instant, `value.key` would point to a non-existent address, which is not legal. To fix this, the entries of `WeakMap` and `WeakSet` aren't actual references, but ephemerons, an enhancement to the mark-and-sweep mechanism. Barros et al. offers a good summary of the algorithm (page 4). To quote a paragraph:

> Ephemerons are a refinement of weak pairs where neither the key nor the value can be classified as weak or strong. The connectivity of the key determines the connectivity of the value, but the connectivity of the value does not affect the connectivity of the key. […] when the garbage collection offers support to ephemerons, it occurs in three phases instead of two (mark and sweep).

As a rough mental model, think of a `WeakMap` as the following implementation:

> ⚠ **Warning:** This is not a polyfill nor is anywhere close to how it's implemented in the engine (which hooks into the garbage collection mechanism).

JS

```js
class MyWeakMap {
  #marker = Symbol("MyWeakMapData");
  get(key) {
    return key[this.#marker];
  }
  set(key, value) {
    key[this.#marker] = value;
  }
  has(key) {
    return this.#marker in key;
  }
  delete(key) {
    delete key[this.#marker];
  }
}
```

As you can see, the `MyWeakMap` never actually holds a collection of keys. It simply adds metadata to each object being passed in. The object is then garbage-collectable via mark-and-sweep. Therefore, it's not possible to iterate over the keys in a `WeakMap`, nor clear the `WeakMap` (as that also relies on the knowledge of the entire key collection).

For more information on their APIs, see the keyed collections guide.

WeakRefs and FinalizationRegistry

> **Note:** `WeakRef` and `FinalizationRegistry` offer direct introspection into the garbage collection machinery. Avoid using them where possible because the runtime semantics are almost completely unguaranteed.

All variables with an object as value are references to that object. However, such references are *strong* — their existence would prevent the garbage collector from marking the object as eligible for collection. A `WeakRef` is a *weak reference* to an object that allows the object to be garbage collected, while still retaining the ability to read the object's content during its lifetime.

One use case for `WeakRef` is a cache system which maps string URLs to large objects. We cannot use a `WeakMap` for this purpose, because `WeakMap` objects have their *keys* weakly held, but not their *values* — if you access a key, you would always deterministically get the value (since having access to the key means it's still alive). Here, we are okay to get `undefined` for a key (if the corresponding value is no longer alive) since we can just re-compute it, but we don't want unreachable objects to stay in the cache. In this case, we can use a normal `Map`, but with each value being a `WeakRef` of the object instead of the actual object value.

JS

```js
function cached(getter) {
  // A Map from string URLs to WeakRefs of results
  const cache = new Map();
  return async (key) => {
    if (cache.has(key)) {
      const dereferencedValue = cache.get(key).deref();
      if (dereferencedValue !== undefined) {
        return dereferencedValue;
      }
    }
    const value = await getter(key);
    cache.set(key, new WeakRef(value));
    return value;
  };
}

const getImage = cached((url) => fetch(url).then((res) => res.blob()));
```

`FinalizationRegistry` provides an even stronger mechanism to observe garbage collection. It allows you to register objects and be notified when they are garbage collected. For example, for the cache system exemplified above, even when the blobs themselves are free for collection, the `WeakRef` objects that hold them are not — and over time, the `Map` may accumulate a lot of useless entries. Using a `FinalizationRegistry` allows one to perform cleanup in this case.

JS

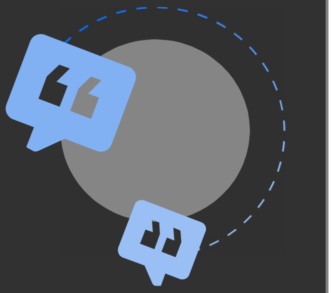
```js
function cached(getter) {
  // A Map from string URLs to WeakRefs of results
  const cache = new Map();
  // Every time after a value is garbage collected, the callback is
  // called with the key in the cache as argument, allowing us to remove
  // the cache entry
  const registry = new FinalizationRegistry((key) => {
    // Note: it's important to test that the WeakRef is indeed empty.
    // Otherwise, the callback may be called after a new object has been
    // added with this key, and that new, alive object gets deleted
    if (!cache.get(key)?.deref()) {
      cache.delete(key);
    }
  });
  return async (key) => {
    if (cache.has(key)) {
      return cache.get(key).deref();
    }
    const value = await getter(key);
    cache.set(key, new WeakRef(value));
    registry.register(value, key);
    return value;
  };
}

const getImage = cached((url) => fetch(url).then((res) => res.blob()));
```

Due to performance and security concerns, there is no guarantee of when the callback will be called, or if it will be called at all. It should only be used for cleanup — and non-critical cleanup. There are other ways for more deterministic resource management, such as `try...finally`, which will always execute the `finally` block. `WeakRef` and `FinalizationRegistry` exist solely for optimization of memory usage in long-running programs.

For more information on the API of `WeakRef` and `FinalizationRegistry`, see their reference pages.

Help improve MDN



Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Mar 15, 2024 by MDN contributors.