**This is how javascript handles the garbage collection**

Let’s learn how javascript handles the garbage variables?

A horse drawn carriage

Description automatically generated with low confidence

<https://blog-assets.risingstack.com/>

For a good program, memory management is very important because memory leakage could cause a big problem with efficiency. So as a good javascript programmer we should have knowledge about, how javascript handles the garbage collection.

Memory management in JavaScript is performed automatically and invisibly to us. We create primitives, objects, functions… All that takes memory.

What happens when something is not needed anymore? How does the JavaScript engine discover it and clean it up? Let's explore it.

**Reachability**

The main concept of memory management in JavaScript is *reachability*.

Simply put, “reachable” values are those that are accessible or usable somehow. They are guaranteed to be stored in memory.

1. There’s a base set of inherently reachable values, that cannot be deleted for obvious reasons.

For instance:

* The currently executing function, its local variables, and parameters.
* Other functions on the current chain of nested calls, their local variables, and parameters.
* Global variables.
* (there are some other, internal ones as well)

These values are called *roots*.

1. Any other value is considered reachable if it’s reachable from a root by a reference or by a chain of references.
2. For instance, if there’s an object in a global variable, and that object has a property referencing another object, *that* object is considered reachable. And those that it references are also reachable.

There’s a background process in the JavaScript engine that is called [garbage collector](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)). It monitors all objects and removes those that have become unreachable.

**Examples**

// user has a reference to the object  
let user = {  
 name: "John"  
};

Here the arrow depicts an object reference. The global variable "user" references the object {name: "John"} (we’ll call it John for brevity). The "name" property of John stores a primitive, so it’s painted inside the object.

If the value user is overwritten, the reference is lost:

user = null

Now John becomes unreachable. There’s no way to access it, no references to it. The garbage collector will junk the data and free the memory.

**Algorithm**

The basic garbage collection algorithm is called **mark-and-sweep**.

The following “garbage collection” steps are regularly performed:

* The garbage collector takes roots and “marks” (remembers) them.
* Then it visits and “marks” all references from them.
* Then it visits marked objects and marks *their* references. All visited objects are remembered, so as not to visit the same object twice in the future.
* And so on until every reachable (from the roots) reference is visited.
* All objects except marked ones are removed.

For instance, let our object structure look like this:

Diagram

Description automatically generated

Now the objects that could not be visited in the process are considered unreachable and will be removed:

Diagram

Description automatically generated

Some of the optimizations:

* **Generational collection** — objects are split into two sets: “new ones” and “old ones”. Many objects appear, do their job, and die fast, they can be cleaned up aggressively. Those that survive for long enough, become “old” and are examined less often.
* **Incremental collection** — if there are many objects, and we try to walk and mark the whole object set at once, it may take some time and introduce visible delays in the execution. So the engine tries to split the garbage collection into pieces. Then the pieces are executed one by one, separately. That requires some extra bookkeeping between them to track changes, but we have many tiny delays instead of a big ones.
* **Idle-time collection** — the garbage collector tries to run only while the CPU is idle, to reduce the possible effect on the execution.

Thanks for reading. I hope it helps you to understand the concept of how javascript handles the garbage collection🙏✌️.