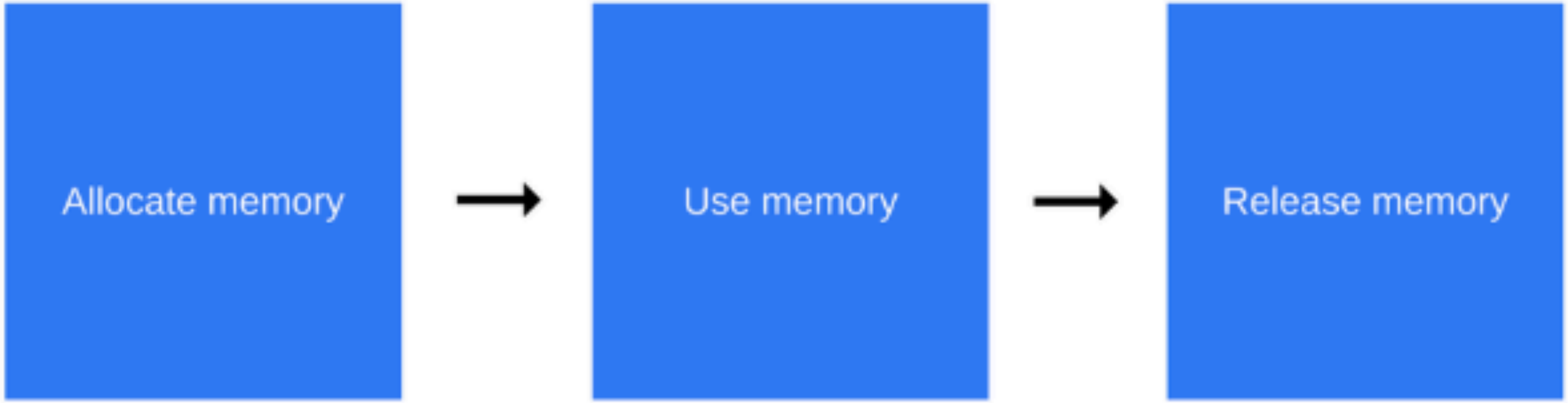
MEMORY MANAGEMENT. GARBAGE COLLECTOR.

by Anastasiia Derkach

softserve

MEMORY MANAGEMENT



Allocate memory

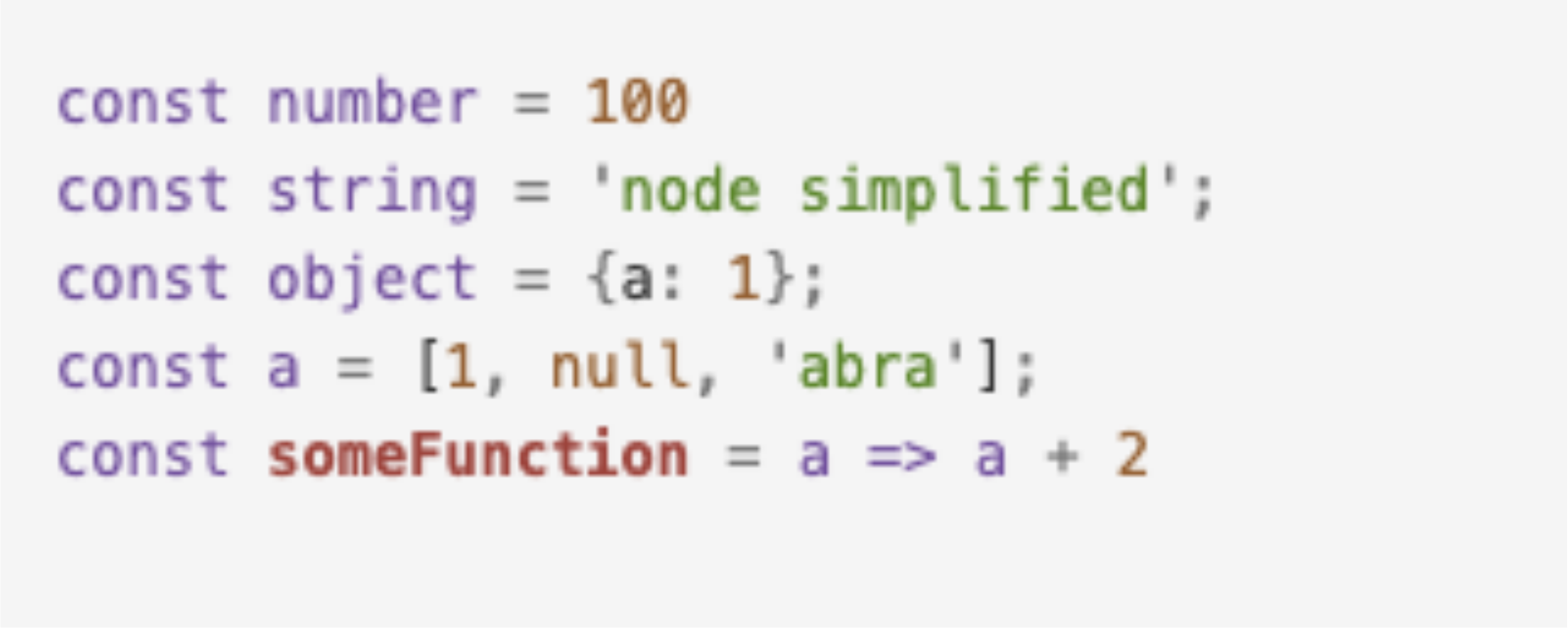
Use memory

Release memory

ALLOCATE MEMORY

Allocate memory — memory is allocated by the operating system which allows your program to use it. In low-level languages (e.g. C) this is an explicit operation that you as a developer should handle. In high-level languages, however, this is taken care of for you.

ALLOCATE MEMORY



```
const number = 100
const string = 'node simplified';
const object = {a: 1};
const a = [1, null, 'abra'];
const someFunction = a => a + 2
```

USE MEMORY

Use memory — this is the time when your program actually makes use of the previously allocated memory. Read and write operations are taking place as you’re using the allocated variables in your code.

RELEASE MEMORY

Release memory — now is the time to release the entire memory that you don’t need so that it can become free and available again. As with the Allocate memory operation, this one is explicit in low-level languages.

BASIC CONCEPT

- Heap memory
- Garbage collection
- Memory leak
- Memory graph

HEAP MEMORY

The heap area is the dynamic memory pool where objects are allocated when they are created by a program.

GARBAGE COLLECTOR

The garbage collector is the process that frees a memory area when the objects allocated in it, are no longer retained (being used) by the program.

MEMORY LEAK

A memory leak is a memory area that cannot be collected even when it will be no longer actively used.

MEMORY GRAPH

The memory graph is the visual representation of how memor is organized, showing variable values starting in the root node until leaf nodes

GARBAGE COLLECTOR

GARBAGE COLLECTION ALGORITHMS

Reference-counting garbage collection

Mark-and-sweep algorithm

REFERENCE-COUNTING

This algorithm looks out for those objects which have no references left. An object becomes eligible for garbage collection if it has no references attached to it.

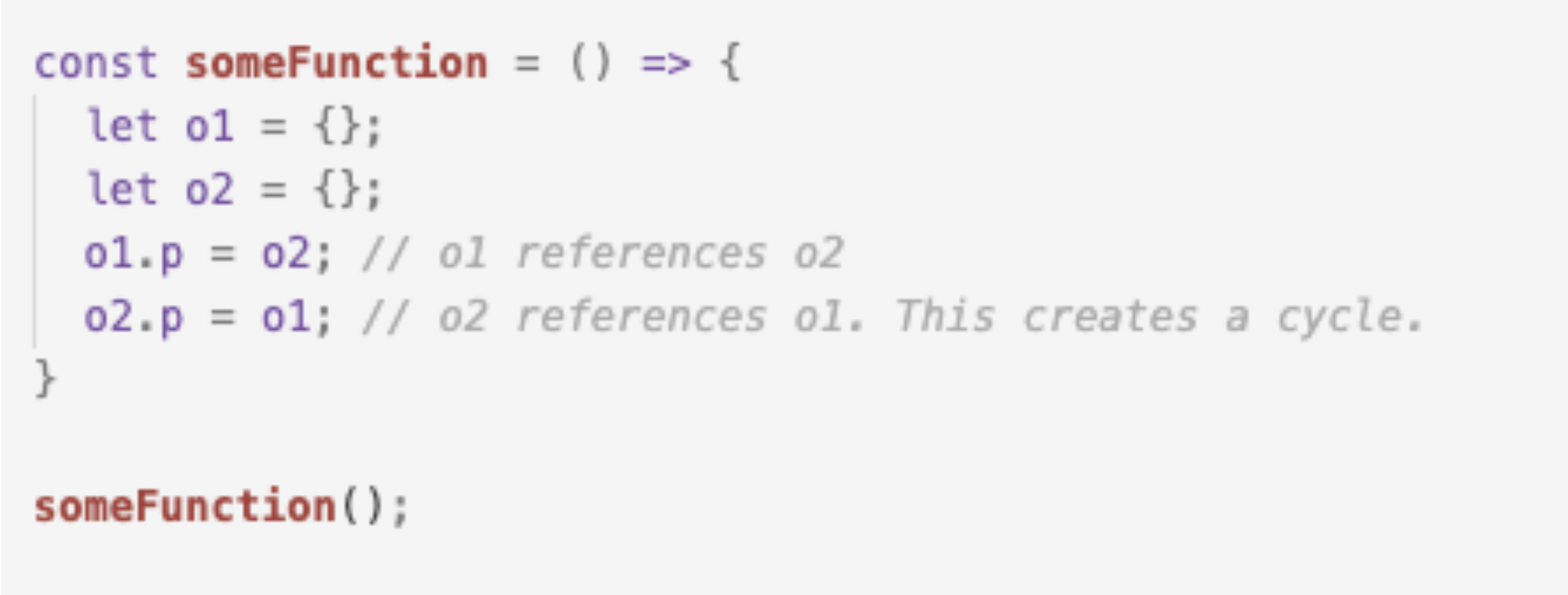
1. Local variables or a function when it is executed and returns.

2. Any object whose all references (variables pointing to) have been nullified (or deleted).

3. Closure scope variables when the function references disappear.

4. DOM Nodes where they are not in the DOM tree and there is no variable pointing to it.

CYCLES ARE CREATING PROBLEMS



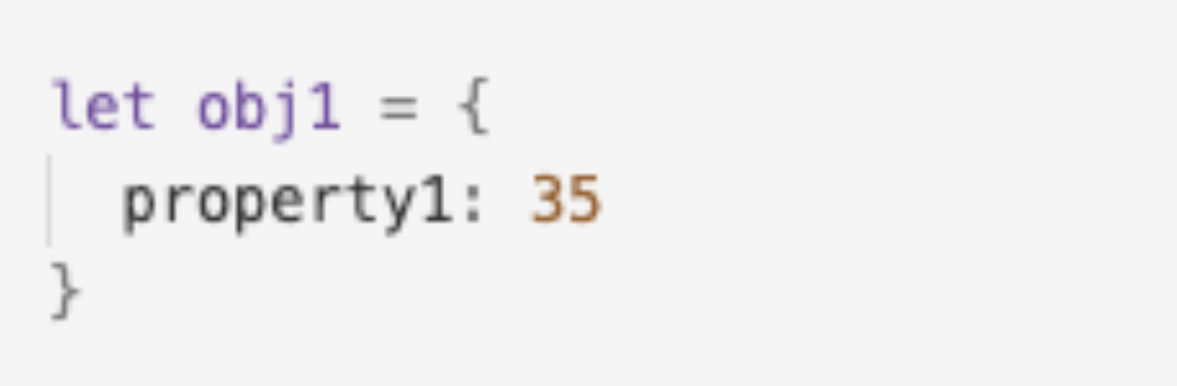
```
const someFunction = () => {
  let o1 = {};
  let o2 = {};
  o1.p = o2; // o1 references o2
  o2.p = o1; // o2 references o1. This creates a cycle.
}

someFunction();
```

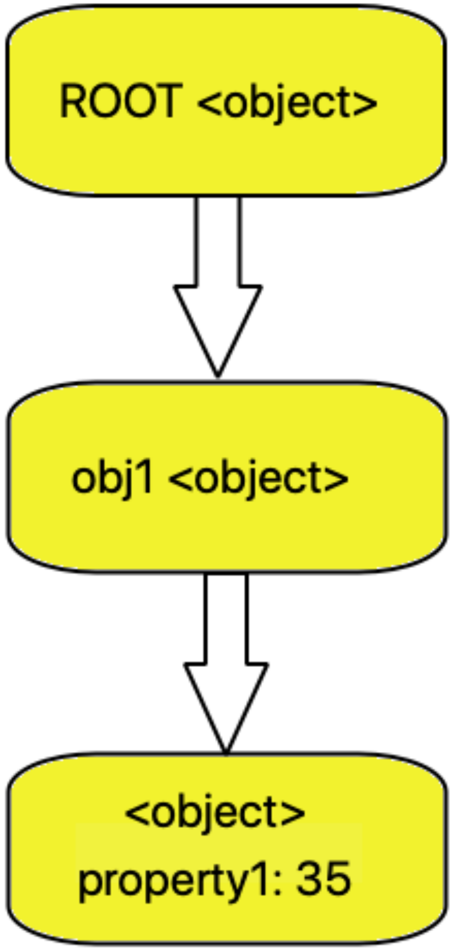
MARK-AND-SWEEP ALGORITHM

This algorithm looks out for objects which are unreachable from the root which is the JavaScript’s global object.

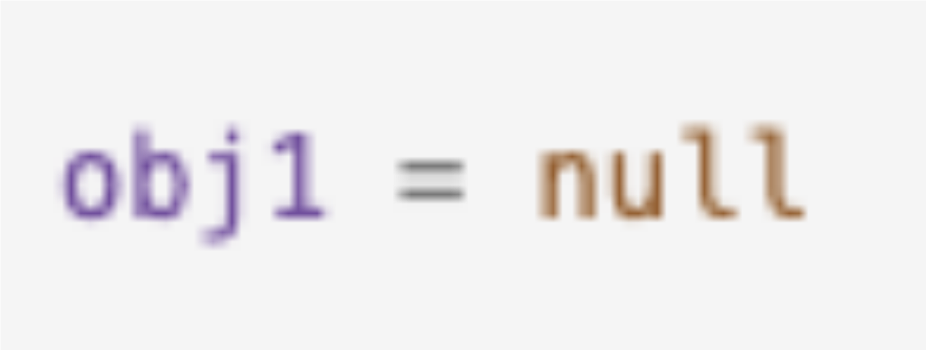
MARK-AND-SWEEP ALGORITHM



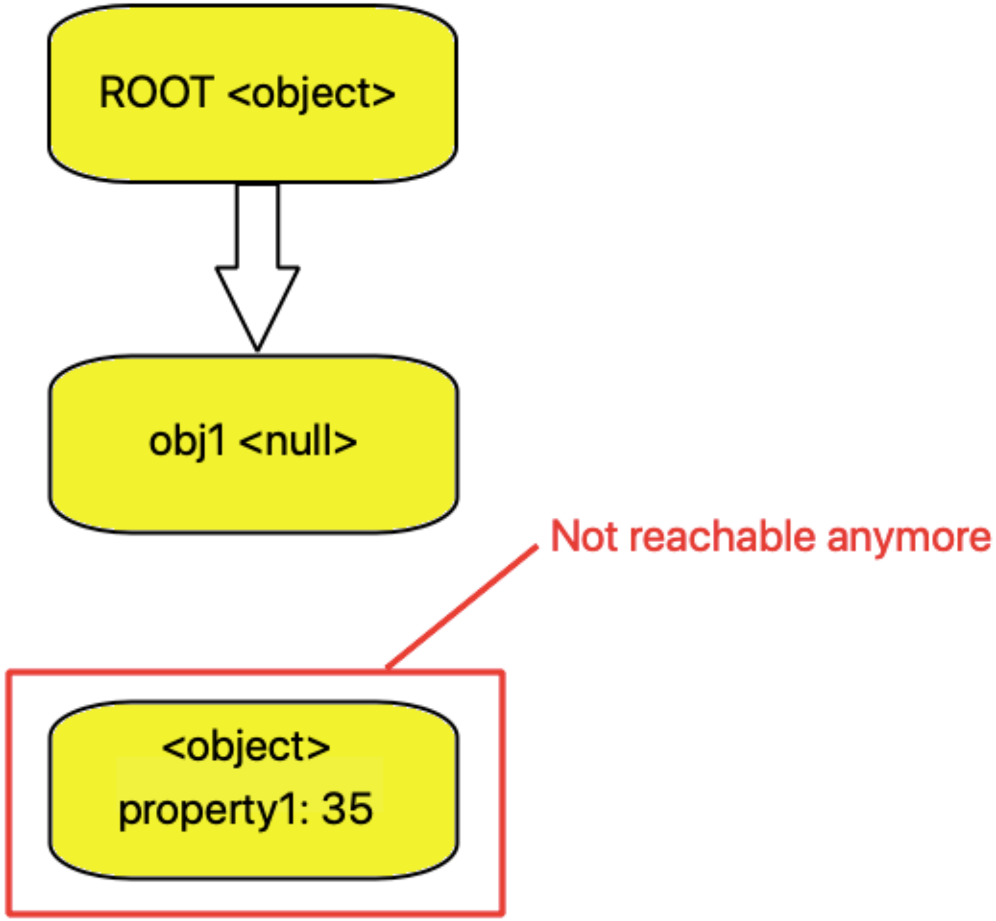
```
let obj1 = {
  property1: 35
}
```



MARK-AND-SWEEP ALGORITHM

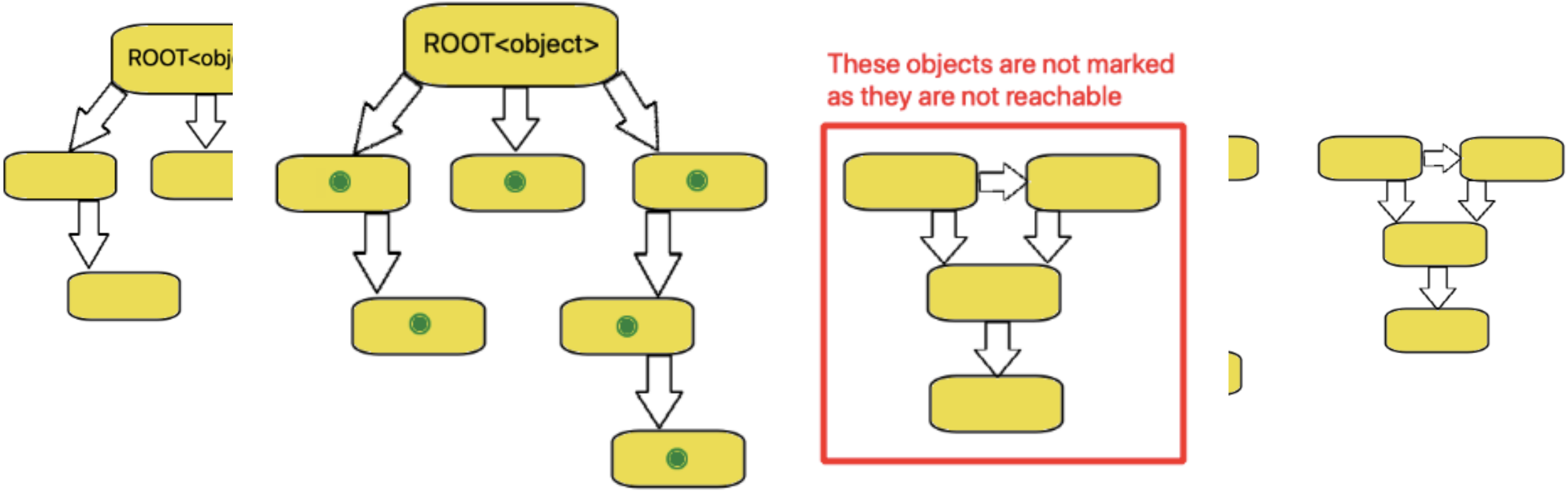


```
obj1 = null
```



softserve

MARK-AND-SWEEP ALGORITHM



These objects are not marked as they are not reachable

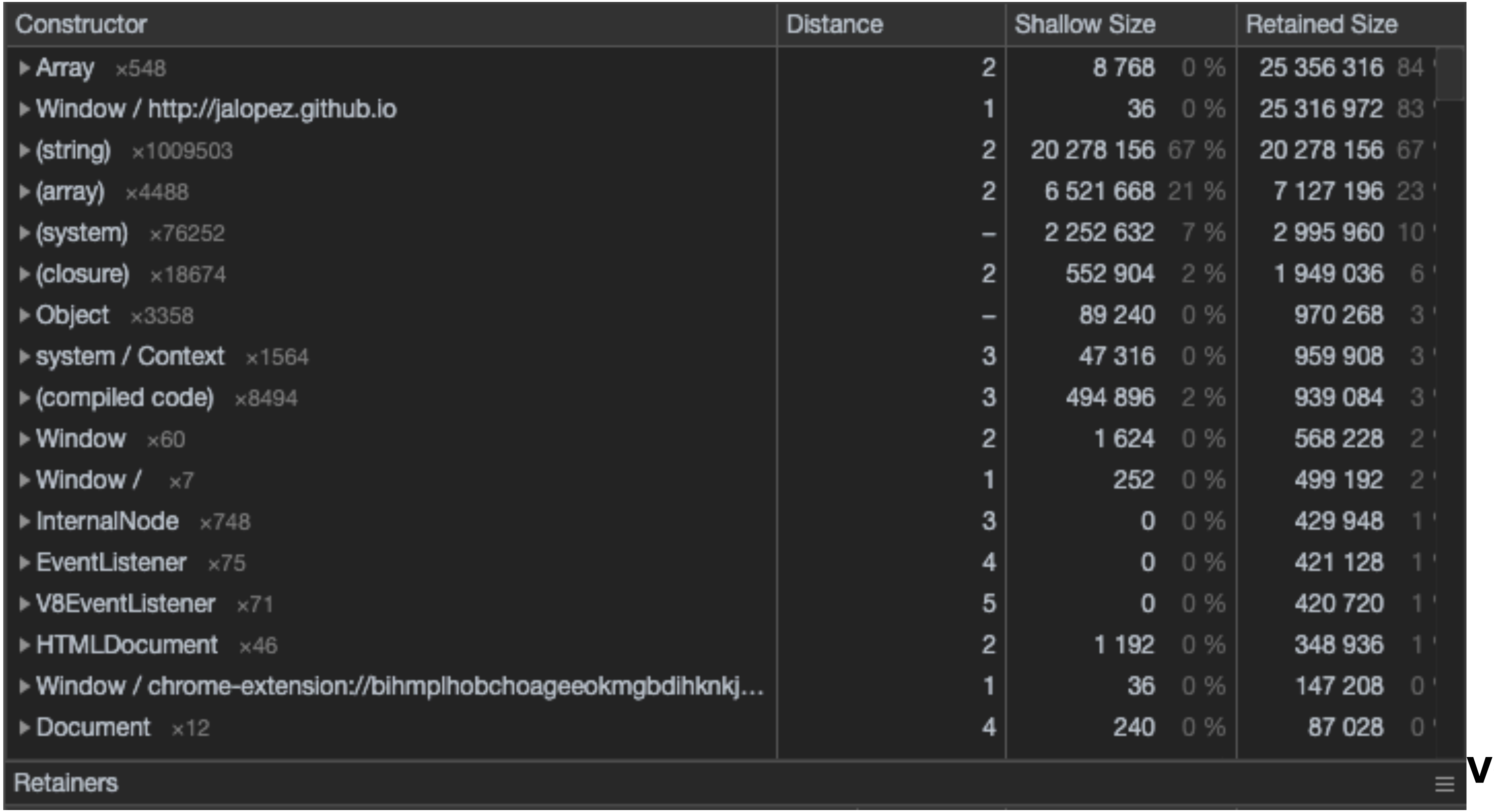
MEMORY LEAK

1. Circular references in objects/functions.
2. "Global" variables (lists, etc.) never emptied.
3. DOM nodes not removed from JS objects.

MEMORY LEAK. FINDING AND KILLING.

1. Check initial memory state.
2. Execute the suspicious action that should not increase memory.
3. Check memory again.
4. If memory is significantly higher, There is a leak.
5. Use Profiles and Timeline to track what is happening:

- Amount of DOM nodes;
- Retained memory



Constructor	Distance	Shallow Size	Retained Size
▶ Array ×548	2	8 768 0 %	25 356 316 84
▶ Window / http://jalopez.github.io	1	36 0 %	25 316 972 83
▶ (string) ×1009503	2	20 278 156 67 %	20 278 156 67
▶ (array) ×4488	2	6 521 668 21 %	7 127 196 23
▶ (system) ×76252	–	2 252 632 7 %	2 995 960 10
▶ (closure) ×18674	2	552 904 2 %	1 949 036 6
▶ Object ×3358	–	89 240 0 %	970 268 3
▶ system / Context ×1564	3	47 316 0 %	959 908 3
▶ (compiled code) ×8494	3	494 896 2 %	939 084 3
▶ Window ×60	2	1 624 0 %	568 228 2
▶ Window / ×7	1	252 0 %	499 192 2
▶ InternalNode ×748	3	0 0 %	429 948 1
▶ EventListener ×75	4	0 0 %	421 128 1
▶ V8EventListener ×71	5	0 0 %	420 720 1
▶ HTMLDocument ×46	2	1 192 0 %	348 936 1
▶ Window / chrome-extension://bihmplhobchoageeokmgbdihknkj…	1	36 0 %	147 208 0
▶ Document ×12	4	240 0 %	87 028 0

Retainers

ve

1. Distance: number of jumps from the root (window).
2. Shallow size: size of the object itself.
3. Retained size: size of the object plus the objects it is referencing.
4. Retainers: list of objects retaining this object/node.

- Yellow Node has a JavaScript reference;
- Red Node referenced by a yellow node;

1. Don’t rely too much on Internet libraries
2. Don’t keep DOM references in code when don′t need them
3. The less global state, the better
4. Take care about scope
5. Use data caches

RESOURCES USED

- https://deepu.tech/memory-management-in-programming/
- https://www.cronj.com/blog/memory-management-javascript/
- https://blog.sessionstack.com/how-javascript-works-memory-management-how-to-handle-4-common-memory-leaks-3f28b94cfbec
- https://medium.com/front-end-weekly/understanding-javascript-memory-management-using-garbage-collection-35ed4954a67f