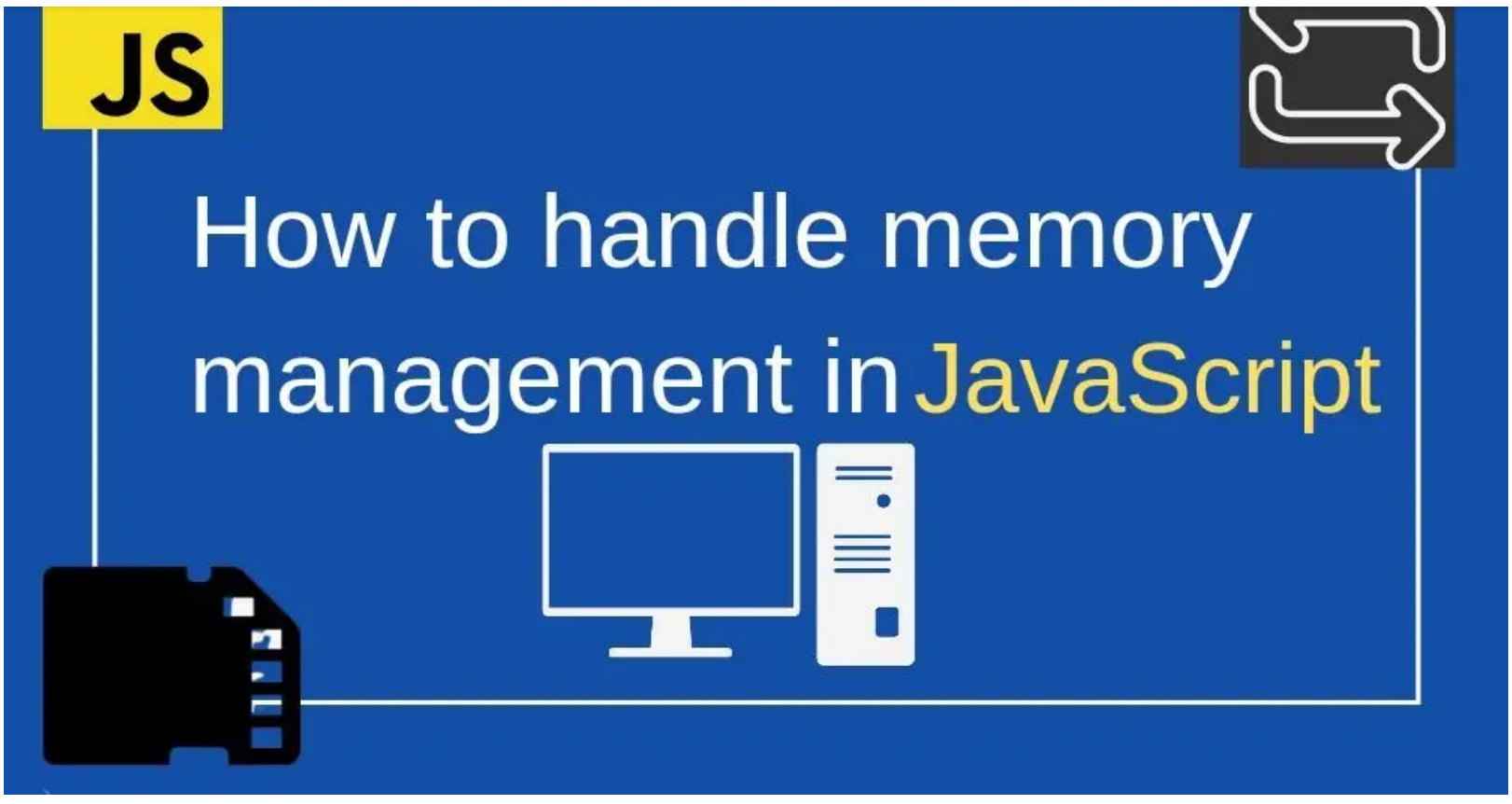


When a user visits a website or an application, two things matter to the user:

1. How long it will take the application to open.
2. The amount of memory space that will be consumed by the program.

Every user who visits an application deserves a rich and interactive experience. They want an application to multitask with other applications that are open on its system simultaneously. This can be done by proper memory management in your program.

Let's find out more about memory management and more precisely about JavaScript memory management and how to handle memory management in JavaScript.

Table of Contents

JavaScript Memory Management

Unlike C language which uses malloc() and free() to allocate and free up memory respectively, is a manual memory management system. JavaScript automatically takes care of the process. This is why JavaScript is a garbage-collected language.

JavaScript removes the pain of memory management by automatically allocating its memory and freeing it up (garbage collection) when not in use. However, most JavaScript developers don't bother themselves about JavaScript memory management.

This is important to know because if we know how JavaScript allocates its memory, then we will be able to use the memory optimally and effectively. Again if we understand how JavaScript runs the garbage collection algorithm, then we can settle some of the memory leakages which may arise.

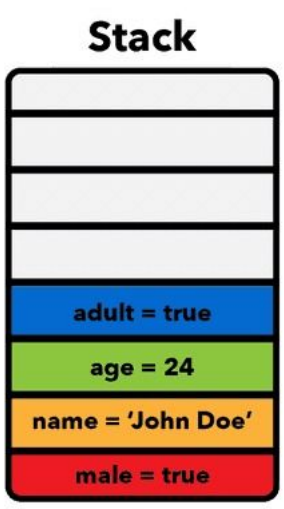
What is memory management?

The next question is where does JavaScript store this data?

JavaScript engines store their data in two places; the Stack Memory and the Heap Memory.

1. Stack Memory - Static Memory allocation: is a type of data structure that uses the Last-in-First-out (LIFO) manner, to store static data. Because of its fixed size, known during compile time by the engine, it is static. Static data in JavaScript comprises references to objects and functions as well as primitive values such as "strings, number, Boolean, null, undefined, symbol, BigInt."

```
const male = true
const name = 'John Doe'
const age = 24
const adult = true
```



It allocates a set amount of memory for every value because it has a defined size that won't change.

2. Heap – Dynamic memory allocation: Heap is another way of storing data in memory. This is used for storing objects (in this context here, our objects mean both object and functions) in memory.

The JavaScript heap doesn't allocate a fixed amount of memory like the stack does, instead, it allocates more space during run time i.e the size is known at run time and there is no limit for its object memory.

Here is a summary of the two storage:

Stack	Heap
Primitive data type and values	Objects, functions and arrays
Size is known at compile time	Size is known at run time
Allocated a fixed amount of memory	No limit to the amount of memory

Let's look at a few code examples for easy understanding.

```
//Arrays are object.It is stored in heap.

let myName = "Joshua"; // allocates memory for a string
const age = 100; // allocates memory for a number
myName = "SirJosh"; // allocates memory for a new string

// Primitives are immutable.Thus,new memory is allocated when new value is assigned.
```

References in JavaScript memory management

All variables start by pointing to the stack. A reference to the item in the heap is stored in the stack if the value is not primitive.

We need to maintain a reference to the heap in the stack since the memory of the JavaScript heap is not organized in any specific way. The objects in the heap can be compared to residences, with references serving as the references' addresses.

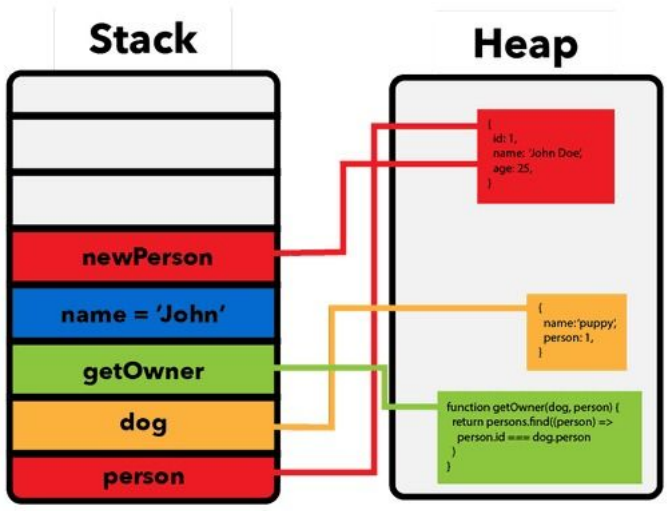
```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
     person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```



From the picture we can see how different values are stored, both the **“person”** and **“newPerson”** objects are stored in the heap and they point to the same object (our object also means object in JS and functions). But a reference to it is stored in the stack.

JavaScript garbage collection

We have known how the allocation of memory works. Where JavaScript stores its memory. But the memory life cycle which we discussed previously, shows that there is one last step; releasing the memory when not in use. This process is handled automatically by JavaScript i.e the JavaScript garbage collector takes care of this.

The moment the JavaScript engine realizes that a variable, object, function, or array is no longer required. It liberates the memory that it takes up. However, how can we determine when these are no longer required? It is quite tough to predict it precisely, however several techniques (algorithms) assist us to come up with a solid solution.

Reference-Counting Garbage Collection in JavaScript

This means removing memory from the heap that has no reference to them in the stack.

We have an object person and in that person object, we have an array (arrays are objects in JavaScript). Both the newPerson and the Person is pointing to the red cycle in the heap i.e making reference to the red cycle in the heap. Later, we created hobbies variable, holding the hobbies array in the person object and this hobbies variable is stored in the stack while the value is stored in the JavaScript heap (since it is an object).

When we intentionally set person and newPerson to null (intentional absence of any object value). The reference counting garbage checks whether an object on the heap has a reference pointing to it from the stack, if no references are pointed then it removes those objects from the heap, leaving only the array(objects) that has a reference it is pointing to.

The issue with them is that they don't understand the cyclic reference, or when two objects are referencing one another.

Cyclic reference problem

A cyclic reference problem occurs when both objects are referencing each other.

```
let boy = {
  name: "Joshua",
};

let hobbies = {
  name: "Football",
};

//cyclic referencing
boy.hobbies = hobbies;
hobbies.boy = boy;

boy = null;
hobbies = null;
```

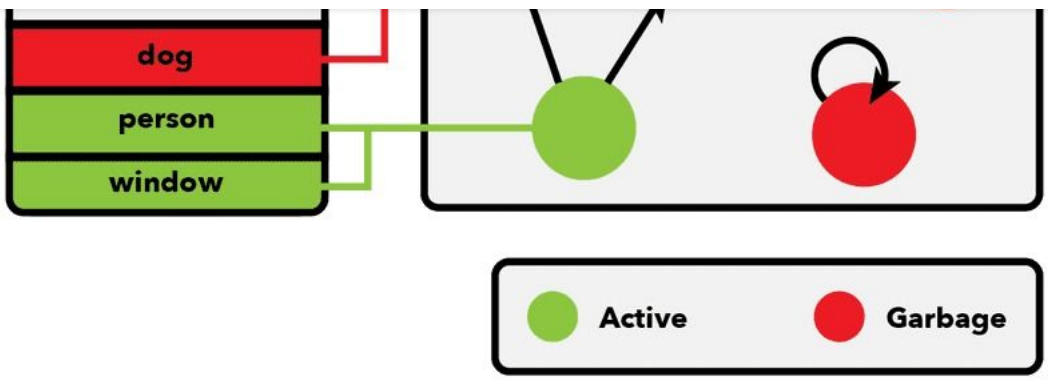
The boy and hobbies are both references to one another in the code above, therefore the algorithm won't free the memory it has been given.

Setting **boy** and **hobbies** to null won’t make the reference-counting collection algorithm recognize what is going on, because both of them have incoming references on the heap.

Mark and Sweep Algorithm in JavaScript

It works almost the same way as the Reference-Counting Algorithm, just that it resolves the cyclic reference problem. Mark and sweep algorithm checks if a variable, object or array is reachable from the root object and not a reference to a particular object.

Note: The root is the window object in JavaScript while it is the global object in NodeJs.



The Mark and Sweep Algorithm mark objects that can't be reached from the root object as garbage and sweep (collect) them off. In our last example, both the boy and hobbies object will be swept (collected) away because they are not reachable from the root object. So, it is classified as garbage. Root objects are not collected.

Important Note

Automatic collected language (e.g JavaScript) allows us to focus on building applications without worrying about memory management. However, there are some side effects as well.

The Stop-The-World garbage collection technique means to halt the program and execute the garbage collection cycle. This means that JavaScript garbage collection runs periodically and JavaScript developers don't know when it will happen, and if it happens frequently it will affect the performance of the application and due to the Stop-The-World technique, the JavaScript program and application are likely to use more memory than they need.

Memory leaks (JavaScript memory leaks)

A memory leak, is a memory allocation that the JavaScript engine is unable to recover. When you add objects and variables to your program, the JavaScript engine allocates memory, and it is intelligent enough to release the memory when the objects are no longer required. Logic errors lead to memory leaks, which negatively impact the speed of your program.

Being familiar with what JavaScript memory management is and what JavaScript memory leaks means. Let's look at the most common JavaScript memory leaks.

Global Variables

When data is being stored in global variables, it causes memory leaks e.g the use of var in your code instead of let or const, also undeclared variables are being stored in the global object.

```
name = "Joshua"; //undeclared variable
var nickName = "SirJosh"; //use of var
```

Both codes are stored in the global object and it can be accessed by window.name and window.nickName.

Also, Declaration functions are stored in the global scope(the window object).

To avoid this “use strict” mode to enable stricter and more secure applications and also prevent unwanted global variables or you can assign the global variable to null (i.e window.name = null) after use to prevent JavaScript memory leaks because such references are directly stored in the root and cannot be collected.

Closure

According to MDN source“A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment)”. In simple terms, it is when a nested function has access to its parent function.

Variables that are scoped by a function are cleaned up once the function has left the call stack, whereas variables that are scoped by a closure are still referenced after the function has finished running. Unused outer scope variables are stored in memory, hence this is a common reason for memory leaks.

```
function outerFilm() {
  let movie = "who am i";

  function innerFilm() {
    let movie = "rush hour";
    console.log(movie.toUpperCase());
  }
  inner();
}
```

In the example above, outerFilm() is never returned and cannot be reached by the garbage collector, thereby increasing its size through repeated calls. To resolve this, make sure the variables in the outer scope are either used or returned because closures are inevitable.

Forgotten timers

SetTimeout and SetInterval are two timing events available in JavaScript that are very important features.

SetTimeout is an asynchronous function that executes after the set time (given time) usually in milliseconds expires, while SetInterval allows repeated execution of a code at different intervals (set time). The majority of memory leaks are caused by these timers.

```
const program = {};
const differentInterval = setInterval(function () {
  // everything here won't be collected until the interval is cleared
  doSomething(program);
}, 2000);
```

```
let father = document.getElementById("#father");
let son = document.getElementById("#son");

father.addEventListener("click", function () {
  //here, it remove son from the DOM, but not from the object memory
  son.remove();
});
```

As the event listener is constantly active and contains the son reference; even after the son element was deleted from the DOM, in the code above upon the father clicks, the son variable continues to hold memory. The reason is the garbage collector is unable to release the son object and will keep using memory.

When an event listener is no longer required, you should always unregister it by generating a reference for it and providing it to the removeEventListener method:

```
function removeSon() {
  son.remove();
}
father.addEventListener("click", son);
// after completing required action, remove it using
 "removeEventListener" to avoid memory leaks
father.removeEventListener("click", son);
```

Conclusion

In this content, we learnt about JavaScript memory management, JavaScript memory leaks, the problem they can cause, and how to prevent them.

Memory leaks are caused due to flaws in our code, following the instructions listed above to avoid possible leaks can greatly improve your application and save your memory.

Author

Onwuemene Joshua

Joshua is a frontend developer, a WordPress developer and a Technical writer. He has collaborated on projects which required his Html, CSS/SASS, TailwindCSS and JavaScript skills. He writes on frontend development explaining difficult concepts in a beginner-friendly manner.

Related articles

How to Use JavaScript for Backend Development in 2023

JavaScript is a programming language that allows you to create dynamically updated...

Read more

How to Build Routes in Flask

Curious to learn how to build your own Flask routes to create dynamic web applications...

Read more

How to Validate URLs in JavaScript

Ever wondered how to validate URLs in JavaScript? The process is surprisingly simple...

Read more

Apply for remote JavaScript developer jobs at top U.S. companies

ıgineer

: is providing
ls to
:onsultations,
ıt services, is
:nd Engineer.
ısible for...

mployees

FULL-TIME REMOTE JOB

Senior Full-Stack Developer

A U.S.-based company that is on a mission to revolutionize the existing cyber-security experience by providing customers with innovative integrated and connected solutions, is looking for a Senior Full-Stack Developer. The...

Technology 1-10 employees

JavaScript | Ruby on Rails | GraphQL | + 1

Apply now

FULL-TIME REMOTE JOB

Web Engineer

A U.S.-based company that is developing a productive platform to assist with data analysis, employee pay, top talent hiring, and business management, is looking for a Web Engineer. The engineer will be...

Technology 1-10 employees

WordPress | Elementor | JavaScript | + 3

Apply now

FULL-TIME REMOTE JOB

Full-Stack (FE-heavy) D

A rapidly-growing company tha
mission to help businesses grov
efficiently by utilizing their inno
and high-quality software soluti
looking for a Full-Stack Front-E
Developer. The developer will b

Other 1-10 employees

Python | JavaScript

Apply now

Frequently Asked Questions

- When is a memory considered Garbage?
- Why is it necessary for JavaScript developers to be concerned about memory?
- Why should JavaScript developers care about memory?

View more FAQs

Press

What's up with Turing?
Get the latest news about us here.

Blog

Know more about remote work.
Checkout our blog here.

1900 Embarcadero Road Palo Alto, CA, 94303