**Memory Management**

With JavaScript, the primary memory issues that come up relate to ***memory leaks***, which is when memory that should be released is still in action.

JavaScript has two data structures for memory:

* The heap
* The stack

When you assign values to variables, the JavaScript engine is tasked with figuring out if the value is a primitive or a reference value. The outcome determines if the value is stored in the heap or the stack.

**The Stack**

The stack is used for static storage, where the size of an object is known when the code is compiled. Since the size is known, a fixed amount of data is reserved for the object, and the stack remains ordered. The stack has a finite amount of space provided by the operating system, which you typically only exceed when you have problems in your code, like infinite recursion or memory leaks.

**Primitive values, *references* to non-primitive values, and function call frames are stored in the stack**. You can think of references as a parking space number in a massive (but disordered) parking garage telling JavaScript where to find objects and functions.

**The Heap**

The heap provides dynamic memory allocation at runtime for data types that don’t have a fixed size, like **objects and functions**. These are *reference values* and we keep track of where to find them in the unstructured heap using a fixed-size *reference* in the stack. If you modify an object, you are modifying a reference to the object and not the object itself.

const cat = { name: "Jupiter"}  
const pets = ["Jupiter", "Moshi", "Hercules"]

In the example, **cat** is stored in the heap, a reference to **cat** is stored in the stack, and the property **name** is stored in the stack. The **pets** array is stored in the heap, while a reference to it is stored in the stack.

let object = new Object();  
let object2 = object;  
object.greeting = "Hello, world";  
console.log(object2); // { greeting: 'Hello, world' }

In the example, **object** and **object2** are pointing to the same object in memory in the heap, but with different variables that are saved in the stack.

🡪 Objects, arrays are aliasing

**Memory Life Cycle**

1. **Allocation (values declared and stored in memory)**

When we create a variable or declare values, memory is allocated. This can be initiated in many ways:

* Regular variable assignment
* Assigning properties to an object
* Declaring callable functions
* Calling functions

Graphical user interface, text, application

Description automatically generated**2. Memory in use** – When reading and writing allocated memory. Includes variable reassignments, using variables, and passing variables to functions.

**3. Releasing Memory**: Garbage Collection

The JavaScript engine manages garbage collection using two key algorithms:

* Reference-counting
* Mark-and-sweep

Garbage collection algorithms use different approaches to detect if some piece of memory is no longer needed by the program. Once memory is no longer needed, it is released and can be reused.

1. *Reference-counting*

*Reference-counting* makes use of the references to variables that live on the stack. When an object is created, it’s reference count is one. If you make a second variable point to that object, the reference count is two. If a function makes use of an object, the reference count is increased by one. Usually, inner elements from function calls are garbage collected when a function is done executing, unless the inner elements are still referenced.

Text

Description automatically generated

🡪 If reference drops down to 0, JS engine can mark that memory block as free to use.

1. *Mark-and-Sweep*

The *Mark-and-Sweep* algorithm runs periodically and starts at the root of your code, the global object. From the root, it’ll “sweep” across your code to find and mark anything that is “reachable” by traversing across all of the variables. After that process, any of the variables that are unmarked will be garbage collected during the sweep phase. That process is repeated over and over again during code execution.

**Memory Leaks**

There are a few common scenarios that cause memory leaks in code.

1. **Messy Closures**

Text

Description automatically generated

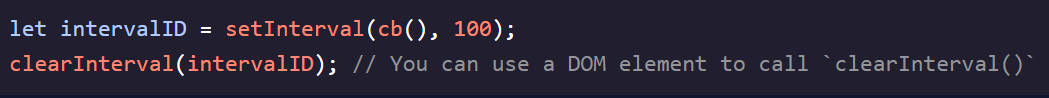
In the example, the closure over the **bigObj** object keeps the memory in use, even after **bigMemoryUser()** finishes executing. If you run this code in your browser, it might crash the browser due to the **console.log()** statements when **bigMemoryUser()** executes 1000 times. The object **bigObj** can grow infinitely depending on how many times **bigMemoryUser()** is called.

1. **Dangling Timers and Event Listeners**

You might be used to using **setInterval()** or other browser APIs in your code. Sometimes, you can wind up with a dangling timer or callback that references nodes or memory that your program doesn’t need anymore. If the handler is still active, anything it is referencing can’t be garbage collected.

Text

Description automatically generatedIn the example, the **counter** variable is in the closure when you call **cb()**. When we use the **setInterval()** callback, it repeatedly calls that function **cb()** every 1000ms (set by the second argument). If you don’t assign the **setInterval()** call to a variable, you’ll get a memory leak if you can’t clear the interval later.



The second snippet shows how you can assign a variable the value of calling **setInterval()** so that you can clear it when the time comes.

Another scenario to watch out for is the existence of anonymous functions when you use event listeners:

Text

Description automatically generated

In the example, the **lotsOfMemory** string will be stored in the closure of the anonymous function that is called on scroll events.

1. **Circular References**

If two objects have pointers that reference each other, a *circular reference* is formed.

Shape

Description automatically generated with low confidence

As you can see in the example, the **first** object has a property **aProperty** that references the **second** object and the **second** object has a property **anotherProperty** that references the **first** object. Since these two objects reference each other through their properties, they’ll each wind up having a reference count of two. Circular references can cause memory leaks due to the reference-counting algorithm. Luckily, the mark-and-sweep algorithm — used by most browsers — handles that shortcoming.

1. **Declaring Variables on the Global Object**

Text

Description automatically generated

This type of issue is easy to avoid as long as you remember scoping rules and always use an appropriate **let** or**const** statement to assign your variables with the correct block scoping. You can also use strict-mode to help keep your global scope clean. Since global variables are available from the root, they never get garbage collected.