Memory Management & Garbage collectors in Java script

INTRODUCTION:

Memory leaks are a problem every developer has to face eventually. Even when working with memory-managed languages there are cases where memory can be leaked. Leaks are the cause of whole class of problems: slowdowns, crashes, high latency, and even problems with other applications.

Memory Management and garbage collection in JavaScript is a slightly unfamiliar topic since in JavaScript we are not performing any memory operations explicitly, however, it is good to know how it works.

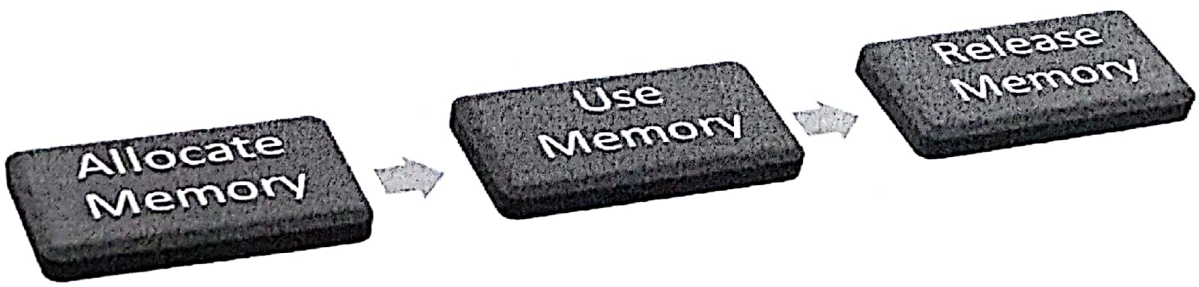
In the low-level languages like C, developers need to manually allocate and deallocate the memory using the malloc(), calloc(), realloc(), and free() methods. In the high-level languages like Java and JavaScript, we don't need to explicitly allocate or release memory. JavaScript values are allocated when things are created (objects, Strings, etc.) and freed automatically when they are no longer used. This process is called Garbage collection.

WHAT ARE MEMORY LEAKS?

In essence, memory leaks can be defined as memory that is not required by an application anymore that for some reason is not returned to the operating system or the pool of free memory. Programming languages favor different ways of managing memory. These ways may reduce the chance of leaking memory. However, whether a certain piece of memory is unused or not is actually an undecidable problem. In other words, only developers can make it clear whether a piece of memory can be returned to the operating system or not. Certain programming languages provide features that help developers do this. Others expect developers to be completely explicit about when a piece of memory is unused.

Memory Life Cycle:

Irrespective of any language (high-level or low-level), memory life cycle will be similar to what is shown below.



In the high-level languages, we will just read and write to the memory explicitly (Use Memory). In the low-level languages, developers need to take care to explicitly perform all three steps.

Since allocation and deallocation happen automatically, that doesn't mean developers don't care about memory management. Poor coding may lead to memory leaks, which is a condition where memory is not released even though it is no longer used by the application. So it is very important to learn more about memory management.

MEMORY MANAGEMENT IN JAVASCRIPT:

JavaScript is one of the so called *garbage collected* languages. Garbage collected languages help developers manage memory by periodically checking which previously allocated pieces of memory can still be "reached" from other parts of the application. In other words, garbage collected languages reduce the problem of managing memory from "what memory is still required?" to "what memory can still be reached from other parts of the application?". The difference is subtle, but important: while only the developer knows whether a piece of allocated memory will be required in the future, unreachable memory can be algorithmically determined and marked for return to the OS.

Garbage Collection:

Garbage collection is the process of finding memory which is no longer used by the application and releasing it. To find the memory which is no longer used, a few algorithms will be used by the garbage collector, and in this section, we will look into the main garbage collection algorithms and their limitations. We will look into following algorithms:

LEAKS IN JAVASCRIPT:

The main cause for leaks in garbage collected languages are *unwanted references*. To understand what unwanted references are, first we need to understand how a garbage collector determines whether a piece of memory can be reached or not.

MARK AND SWEEP:

Most garbage collectors use an algorithm known as *mark-and-sweep*. The algorithm consists of the following steps:

1. The garbage collector builds a list of "roots". Roots usually are global variables to which a reference is kept in code. In JavaScript, the "window" object is an example of a global variable that can act as a root. The window object is always present, so the garbage collector can consider it and all of its children to be always present (i.e. not garbage).

2. All roots are inspected and marked as active (i.e. not garbage). All children are inspected recursively as well. Everything that can be reached from a root is not considered garbage.
3. All pieces of memory not marked as active can now be considered garbage. The collector can now free that memory and return it to the OS.

Modern garbage collectors improve on this algorithm in different ways, but the essence is the same: reachable pieces of memory are marked as such and the rest is considered garbage.

Unwanted references are references to pieces of memory that the developer knows he or she won't be needing anymore but that for some reason are kept inside the tree of an active root. In the context of JavaScript, unwanted references are variables kept somewhere in the code that will not be used anymore and point to a piece of memory that could otherwise be freed. Some would argue these are developer mistakes.

So to understand which are the most common leaks in JavaScript, we need to know in which ways references are commonly forgotten.

The Three Types of Common JavaScript Leaks

1: Accidental global variables

One of the objectives behind JavaScript was to develop a language that looked like Java but was permissive enough to be used by beginners. One of the ways in which JavaScript is permissive is in the way it handles undeclared variables: a reference to an undeclared variable creates a new variable inside the *global*object. In the case of browsers, the global object is window. In other words:

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

Is in fact:

```
function foo(arg) {
window.bar = "this is an explicit global variable";
}
```

If bar was supposed to hold a reference to a variable only inside the scope of the foo function and you forget to use var to declare it, an unexpected global variable is created. In this example, leaking a simple string won't do much harm, but it could certainly be worse.

Another way in which an accidental global variable can be created is through this:

```
function foo() {
   this.variable = "potential accidental global";
}
// Foo called on its own, this points to the global object (window)
// rather than being undefined.
foo();
```

To prevent these mistakes from happening, add 'use strict'; at the beginning of your JavaScript files. This enables a stricter mode of parsing JavaScript that prevents accidental globals.

A NOTE ON GLOBAL VARIABLES

Even though we talk about unsuspected globals, it is still the case that much code is littered with explicit global variables. These are by definition non collectable (unless nulled or reassigned). In particular, global variables used to temporarily store and process big amounts of information are of concern. If you must use a global variable to store lots of data, make sure to null it or reassign it after you are done with it. One common cause for increased memory consumption in connection with globals are caches). Caches store data that is repeatedly used. For this to be efficient, caches must have an upper bound for its size. Caches that grow unbounded can result in high memory consumption because their contents cannot be collected.

2: Forgotten timers or callbacks

The use of setInterval is quite common in JavaScript. Other libraries provide observers and other facilities that take callbacks. Most of these libraries take care of making any references to the callback unreachable after their own instances become unreachable as well. In the case of setInterval, however, code like this is quite common:

```
var someResource = getData();
setInterval(function() {
var node = document.getElementById('Node');
if(node) {
     // Do stuff with node and someResource.
     node.innerHTML = JSON.stringify(someResource));
  }
}, 1000);
```

This example illustrates what can happen with dangling timers: timers that make reference to nodes or data that is no longer required. The object represented by node may be removed in the future, making the whole block inside the interval handler unnecessary. However, the handler, as the interval is still active, cannot be collected (the

interval needs to be stopped for that to happen). If the interval handler cannot be collected, its dependencies cannot be collected either. That means that someResource, which presumably stores sizable data, cannot be collected either.

For the case of observers, it is important to make explicit calls to remove them once they are not needed anymore (or the associated object is about to be made unreachable). In the past, this used to be particularly important as certain browsers (Internet Explorer 6) were not able to manage cyclic references well (see below for more info on that). Nowadays, most browsers can and will collect observer handlers once the observed object becomes unreachable, even if the listener is not explicitly removed. It remains good practice, however, to explicitly remove these observers before the object is disposed. For instance:

```
var element = document.getElementById('button');
function onClick(event) {
  element.innerHtml = 'text';
}
element.addEventListener('click', onClick);
// Do stuff
element.removeEventListener('click', onClick);
element.parentNode.removeChild(element);
// Now when element goes out of scope,
// both element and onClick will be collected even in old browsers that don't
// handle cycles well.
```

A NOTE ABOUT OBJECT OBSERVERS AND CYCLIC REFERENCES

Observers and cyclic references used to be the bane of JavaScript developers. This was the case due to a bug (or design decision) in Internet Explorer's garbage collector. Old versions of Internet Explorer could not detect cyclic references between DOM nodes and JavaScript code. This is typical of an observer, which usually keeps a reference to the observable (as in the example above). In other words, every time an observer was added to a node in Internet Explorer, it resulted in a leak. This is the reason developers started explicitly removing handlers before nodes or nulling references inside observers. Nowadays, modern browsers (including Internet Explorer and Microsoft Edge) use modern garbage collection algorithms that can detect these cycles and deal with them correctly. In other words, it is not strictly necessary to call removeEventListener before making a node unreachable.

Frameworks and libraries such as *jQuery* do remove listeners before disposing of a node (when using their specific APIs for that). This is handled internally by the libraries and makes sure that no leaks are produced, even when run under problematic browsers such as the old Internet Explorer.

3: Out of DOM references

Sometimes it may be useful to store DOM nodes inside data structures. Suppose you want to rapidly update the contents of several rows in a table. It may make sense to store a reference to each DOM row in a dictionary or array. When this happens, two references to the same DOM element are kept: one in the DOM tree and the other in the dictionary. If at some point in the future you decide to remove these rows, you need to make both references unreachable.

```
var elements = {
    button: document.getElementById('button'),
    image: document.getElementById('image'),
    text: document.getElementById('text')
};

function doStuff() {
    image.src = 'http://some.url/image';
    button.click();
    console.log(text.innerHTML);
    // Much more logic
}

function removeButton() {
    // The button is a direct child of body.
    document.body.removeChild(document.getElementById('button'));

    // At this point, we still have a reference to #button in the global
    // elements dictionary. In other words, the button element is still in
    // memory and cannot be collected by the GC.
}
```

An additional consideration for this has to do with references to inner or leaf nodes inside a DOM tree. Suppose you keep a reference to a specific cell of a table (a <td> tag) in your JavaScript code. At some point in the future you decide to remove the table from the DOM but keep the reference to that cell. Intuitively one may suppose the GC will collect everything but that cell. In practice this won't happen: the cell is a child node of that table and children keep references to their parents. In other words, the reference to the table cell from JavaScript code causes the whole table to stay in memory. Consider this carefully when keeping references to DOM elements.

4: Closures

A key aspect of JavaScript development are closures: anonymous functions that capture variables from parent scopes. Meteor developers found a particular casein which due to implementation details of the JavaScript runtime, it is possible to leak memory in a subtle way:

```
var theThing = null;
var replaceThing = function () {
 var originalThing = theThing;
 var unused = function () {
  if (originalThing)
   console.log("hi");
 };
 theThing = {
  longStr: new Array(1000000).join('*'),
  someMethod: function () {
   console.log(someMessage);
  }
 };
};
setInterval(replaceThing, 1000);
```

This snippet does one thing: every time replaceThing is called, theThing gets a new object which contains a big array and a new closure (someMethod). At the same time, the variable unused holds a closure that has a reference to originalThing (theThing from the previous call to replaceThing). Already somewhat confusing, huh? The important thing is that once a scope is created for closures that are in the same parent scope, that scope is shared. In this case, the scope created for the closure someMethod is shared by unused. unused has a reference to originalThing. Even though unused is never used, someMethodcan be used through theThing. And as someMethod shares the closure scope with unused, even though unused is never used, its reference to originalThingforces it to stay active (prevents its collection). When this snippet is run repeatedly a steady increase in memory usage can be observed. This does not get smaller when the GC runs. In essence, a linked list of closures is created (with its root in the form of the theThing variable), and each of these closures' scopes carries an indirect reference to the big array, resulting in a sizable leak.