1. **What is Garbage Collection?**

- When you declare a variable, object, array or function all are stored somewhere in the memory.

  Example:

  ```
  let obj = {
          a: 10
  }
  ```

  (There are multiple locations to hold huge values)

  When the program ends the data inside location is garbage which needs to be cleared.

  In short, clearing these free locations is the process of collecting the garbage or Garbage Collection or GC.
- Garbage collection is a way of managing application memory automatically. The job of the garbage collector (GC) is to reclaim memory occupied by unused objects (garbage).

- **Garbage Collection in JS:**

- JS is a high level language so you do not need to allocate memory. Memory allocation & releasing happens automatically.

  Making the memory free is the process of 'Garbage Collection' & there is a routine that does it, called Garbage Collector.

  GC process is also called automatic memory management with reference to Javascript.
- The garbage collection considers references & it tries to release the memory if a location is not "reachable".

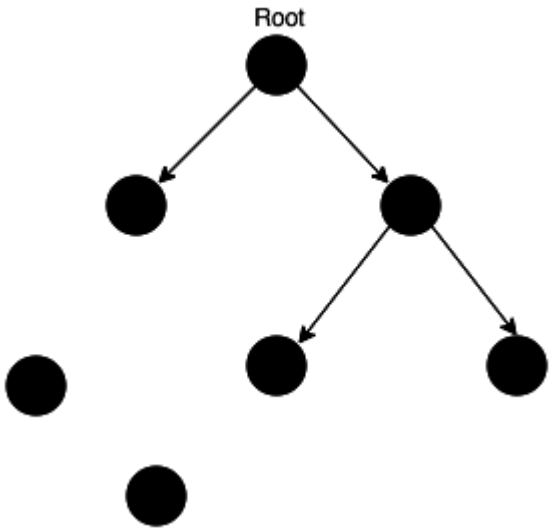
  Example:

  ```
  let obj = {
          a: 10
  }
  let obj = null;
  ```

  Here, obj had reference and then it set to null. Now, nothing can access this value without any reference, so it is not "reachable".

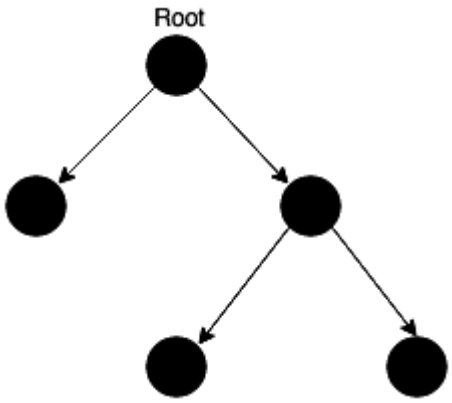
- Memory before Garbage Collection:

The following diagram shows how the memory can look like if you have objects with references to each other, and with some objects that have no reference to any objects. These are the objects that can be collected by a garbage collector run.



- **Memory after Garbage Collection:**

- Once the garbage collector is run, the objects that are unreachable gets deleted, and the memory space is freed up.



- The Advantages of Using a Garbage Collector

- it prevents wild/dangling pointers bugs,
- it won't try to free up space that was already freed up,
- it will protect you from some types of memory leaks.

Things to Keep in Mind When Using a Garbage Collector

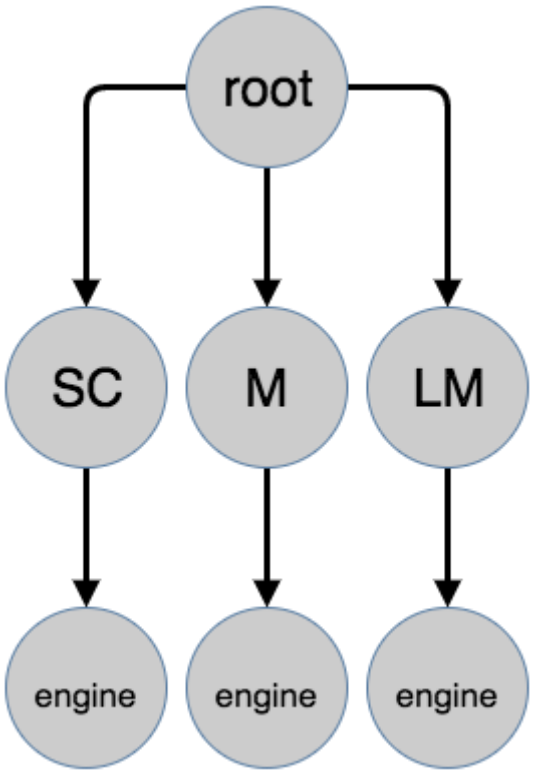
- performance impact - in order to decide what can be freed up, the GC consumes computing power
- unpredictable stalls - modern GC implementations try to avoid "stop-the-world" collections

Node.js Garbage Collection & Memory Management

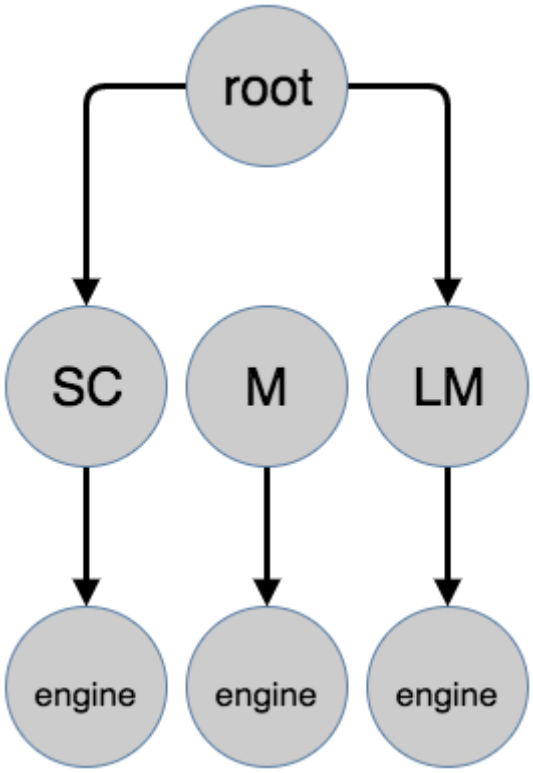
The `Car` object created in the following snippet is placed on the heap.

```
function Engine (power) {
  this.power = power
}

function Car (opts) {
  this.name = opts.name
  this.engine = new Engine(opts.power)
}

let LightningMcQueen = new Car({name: 'Lightning McQueen', power: 900})
let SallyCarrera = new Car({name: 'Sally Carrera', power: 500})
let Mater = new Car({name: 'Mater', power: 100})
```

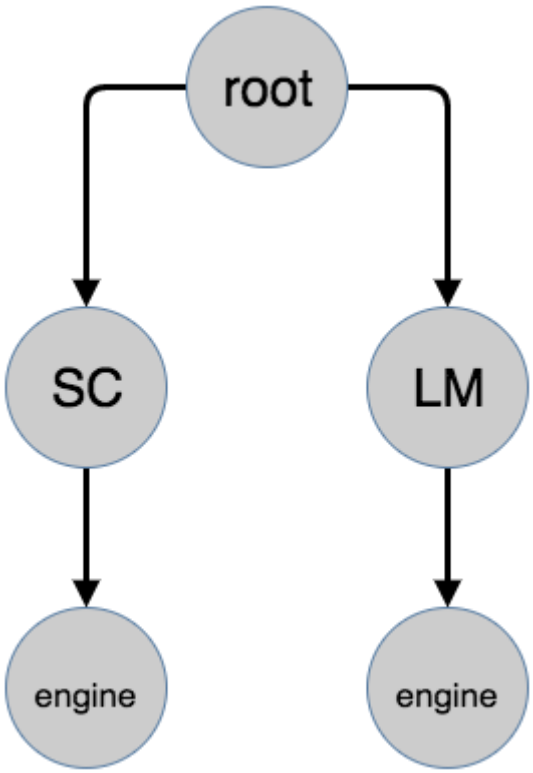
The typical way that closures are implemented is that every function object has a link to a dictionary-style object representing its lexical scope. If both functions defined inside `replaceThing` actually used `originalThing`, it would be important that they both get the same object, even if `originalThing` gets assigned to over and over, so both functions share the same lexical environment. Now, Chrome's V8 JavaScript engine is apparently smart enough to keep variables out of the lexical environment if they aren't used by any closures - from the Meter Blog.



if we no longer use `Mater`, but redefine it and assign some other value, like `Mater = undefined`?



As a result, the original `Mater` object cannot be reached from the root object, so on the next garbage collector run it will be freed up:



Garbage Collection Methods

In one of our previous articles we dealt with how the Node.js garbage collection methods work, so I strongly recommend reading that article.

Here are the most important things you'll learn there:

New Space and Old Space

The heap has two main segments, the New Space and the Old Space. The New Space is where new allocations are happening; it is fast to collect garbage here and has a size of ~1-8MBs. Objects living in the New Space is called Young Generation.

The Old Space where the objects that survived the collector in the New Space are promoted into - they are called the Old Generation. Allocation in the Old Space is fast, however collection is expensive so it is infrequently performed.

Scavenge and Mark-Sweep collection

Scavenge collection is fast and runs on the Young Generation, however the slower Mark-Sweep collection runs on the Old Generation.

A Real-Life Example - The Meteor Case-Study

```
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)
```

2. What are memory leaks?

   In simple terms, a memory leak is nothing but an orphan block of memory on the heap that is no longer used by the application and hasn't been returned to the OS by the garbage collector. So in effect, it's a useless block of memory. An accumulation of such blocks over time could lead to the application not having enough memory to work with or even your OS not having enough memory to allocate, leading to slowdowns and/or crashing of the application or even the OS.

   Memory leak occurs when programmers create a **memory** in heap and forget to delete it. **Memory leaks** are particularly serious **issues** for programs like daemons and servers which by definition never terminate. To avoid **memory leaks**, **memory** allocated on heap should always be freed when no longer needed.

   What causes Memory Leaks in JS:

   Automatic memory management like garbage collection in V8 aims to avoid such memory leaks, for example, circular references are no longer a concern, but could still happen due to unwanted references in the Heap and could be caused by different reasons. Some of the most common reasons are described below.

- Global variables: Since global variables in JavaScript are referenced by the root node (window or global this), they are never garbage collected throughout the lifetime of the application, and will occupy memory as long as the application is running. This applies to any object referenced by the global variables and all their children as well. Having a large graph of objects referenced from the root can lead to a memory leak.
- Multiple References: When the same object is referenced from multiple objects, it might lead to a memory leak when one of the references is left dangling.
- Closures: JavaScript closures have the cool feature of memorizing its surrounding context. When a closure holds a reference to a large object in heap, it keeps the object in memory as long as the closure is in use.

Which means you can easily end up in situations where a closure holding such reference can be improperly used leading to a memory leak.

- Timers & Events: The use of setTimeout, setInterval, Observers and event listeners can cause memory leaks when heavy object references are kept in their callbacks without proper handling.

Best practices to avoid memory leaks

Now that we understand what causes memory leaks, let's see how to avoid them and the best practices to use to ensure efficient memory use.

REDUCE USE OF GLOBAL VARIABLES

Since global variables are never garbage collected, it's best to ensure you don't overuse them. Below are some ways to ensure that.

Avoid Accidental Globals:

When you assign a value to an undeclared variable, JavaScript automatically hoists it as a global variable in default mode. This could be the result of a typo and could lead to a memory leak. Another way could be when assigning a variable to this, which is still a holy grail in JavaScript.

To avoid such surprises, always write JavaScript in strict mode using the `'use strict';` annotation at the top of your JS file. In strict mode, the above will result in an error. When you use ES modules or transpilers like TypeScript or Babel, you don't need it as it's automatically enabled. In recent versions of NodeJS, you can enable strict mode globally by passing the `--use_strict` flag when running the node command.

```
"use strict";

// This will not be hoisted as global variable
function hello() {
    foo = "Message"; // will throw runtime error
}

// This will not become global variable as global functions
// have their own `this` in strict mode
function hello() {
    this.foo = "Message";
}
```

Use Global Scope Sparingly

In general, it's a good practice to avoid using the global scope whenever possible and to also avoid using global variables as much as possible.

1. As much as possible, don't use the global scope. Instead, use local scope inside functions, as those will be garbage collected and memory will be freed. If you have to use a global variable due to some constraints, set the value to null when it's no longer needed.
2. Use global variables only for constants, cache, and reusable singletons. Don't use global variables for the convenience of avoiding passing values around. For sharing data between functions and classes, pass the values around as parameters or object attributes.
3. Don't store big objects in the global scope. If you have to store them, make sure to nullify them when they are not needed. For cache objects, set a handler to clean them up once in a while and don't let them grow indefinitely.

USE STACK MEMORY EFFECTIVELY

Using stack variables as much as possible helps with memory efficiency and performance as stack access is much faster than heap access. This also ensures that we don't accidentally cause memory leaks. Of course, it's not practical to only use static data. In real-world applications, we would have to use lots of objects and dynamic data. But we can follow some tricks to make better use of stack.

```
function outer() {
    const obj = {
        foo: 1,
        bar: "hello",
    };

    const closure = () {
        const { foo } = obj;
        myFunc(foo);
    }
}

function myFunc(foo) {}
```

```
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);
```

The code above creates multiple closures, and those closures hold on to object references. The memory leak, in this case, can be fixed by nullifying originalThing at the end of the replaceThing function. Such cases can also be avoided by creating copies of the object and following the immutable approach mentioned earlier.

When it comes to timers, always remember to pass copies of objects and avoid mutations. Also, clear timers when done, using clearTimeout and clearInterval methods.

The same goes for event listeners and observers. Clear them once the job is done, don’t leave event listeners running forever, especially if they are going to hold on to any object reference from the parent scope.

3. **What is Server-Side Rendering?**

Server-side rendering is the process of taking a client-side only single page application (SPA) and rendering it to static HTML and CSS on the server, on each request. SSR sends a fully rendered page to the client. The client's JS bundle then takes over and the SPA framework can operate as normal.

Why is this important?

- **Improved performance** - The wait time needed to download, parse, and execute the JS code in the browser is eliminated. With static websites or pages, SSR can be useful as it generates the full HTML for a page on the server in response to navigation. This avoids additional round-trips for data fetching and templating on the client since it's taken care of before the browser gets a response, therefore, server-side rendering helps you get your website rendered faster.
- **Faster load times** - SSR generally produces a fast First Paint (FP) and First Contentful Paint (FCP). Faster load times equal a better user experience.
- **Improved SEO** - Another benefit of using SSR is having an app that can be crawled for its content even for crawlers that don't execute JavaScript code. This can help boost SEO.
- **Social sharing** - With SSR, you get a featured image and elaborate snippet when sharing your website's content via social media. This isn't possible for just client-side rendered apps.

Node.js Server Side Rendering (SSR) using EJS

Feature of EJS Module:

1. Use plain javascript.
2. Fast Development time.
3. Simple syntax.
4. Faster execution.
5. Easy Debugging.
6. Active Development.

First of all install express js and ejs using npm install.

npm install ejs

app.js :-

```
// Requiring modules
const express = require('express');
const app = express();
const ejs = require('ejs');
var fs = require('fs');
const port = 8000;

// Render index.ejs file
app.get('/', function (req, res) {

   // Render page using renderFile method
   ejs.renderFile('index.ejs', {},
      {}, function (err, template) {
      if (err) {
         throw err;
      } else {
         res.end(template);
      }
   });
});

// Server setup
app.listen(port, function (error) {
   if (error)
      throw error;
   else
      console.log("Server is running");
});
```

Reference: https://www.geeksforgeeks.org/node-js-server-side-rendering-ssr-using-ejs/

https://blog.appsignal.com/2020/05/06/avoiding-memory-leaks-in-nodejs-best-practices-for-performance.html#:~:text=In%20simple%20terms%2C%20a%20memory,a%20useless%20block%20of%20memory.

https://blog.risingstack.com/node-js-at-scale-node-js-garbage-collection/