1. What happens inside Material UI
2. How event queue works
3. How stack works
4. Real time scenario for closures
5. Demerits of closures
6. HTML 5 New additions
7. Angularjs version
8. Versions of all Technologies
9. HTML session storage and web storage
10. Component did unmount scenario
11. Components without render
12. Why we get undefined instead of null
13. Why componentdidmiunt for API call, will it not create an unending loop?
14. How z index internally works. Stacking Context
15. Top bottom offset do not work with static positioned elements it only works with sticky, relative, fixed, absolute
16. New features in ES6
17. Prototypes, their application

**NodeJS**

NodeJS is a JS runtime, not a language. JavaScript is the language. JavaScript, at least at of the moment of writing, is single-threaded. However, NodeJS has an event-loop and can offload tasks to it's internal thread pool or the Kernel, so that it can execute code while it waits for IO - which allows NodeJS to handle many requests simultaneously, even though it is single-threaded.

**Closures**

JavaScript’s support for closures allow you to register callbacks that, when executed, maintain access to the environment in which they were created even though the execution of the callback creates a new call stack entirely.

**Zero Delay**

Zero delay doesn't actually mean the call back will fire-off after zero milliseconds. Calling [setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setTimeout" \o "The documentation about this has not yet been written; please consider contributing!) with a delay of 0 (zero) milliseconds doesn't execute the callback function after the given interval.

A few useful facts might help clarify what's happening:

1. JavaScript is single-threaded. Asynchronous callbacks are assigned to a *message* placed in a *message queue*.
2. When no code is currently executing, the *event loop* polls the message queue, requesting the next message in line to be processed (executed).
3. setTimeout adds a message (with the callback provided) to the end of this queue after the specified delay has elapsed.

(Note: this means the delay in a setTimeout call is not a sure thing; it is the **minimum delay** before the callback is executed. The actual time taken depends on how long it takes to process any messages ahead of it in the queue.)

So what happens if the delay is set to 0? A new message is added to the queue immediately, and will be processed when the currently executing code is finished and any previously-added messages have been processed.

**What's happening in your code**

**Example 1:**

When you invoke setTimeout…

setTimeout(function() {

console.log('AAA');

});

// has a default time value of 0

…a message gets added to the queue with the specified callback. The rest of your code…

for (i = 0; i < 1000; i++) {

console.log('BBB');

}

// etc.

…continues executing synchronously. Once it has completely finished, the event loop polls the message queue for the next message and finds the one with your setTimeout callback, which is then processed (the callback is run).

The callback only ever gets executed *after* the currently executing code has finished, no matter how long that takes.

**Example 2:**

(function() {

console.log('this is the start');

setTimeout(function cb() {

console.log('Callback 1: this is a msg from call back');

}); // has a default time value of 0

console.log('this is just a message');

setTimeout(function cb1() {

console.log('Callback 2: this is a msg from call back');

}, 0);

console.log('this is the end');

})();

// "this is the start"

// "this is just a message"

// "this is the end"

// "Callback 1: this is a msg from call back"

// "Callback 2: this is a msg from call back"

**Event Loop**

JavaScript runtimes contain a message queue which stores a list of messages to be processed and their associated callback functions. These messages are queued in response to external events (such as a mouse being clicked or receiving the response to an HTTP request) given a callback function has been provided. If, for example a user were to click a button and no callback function was provided – no message would have been enqueued. In a loop, the queue is polled for the next message (each poll referred to as a “tick”) and when a message is encountered, the callback for that message is executed.

**The call stack**

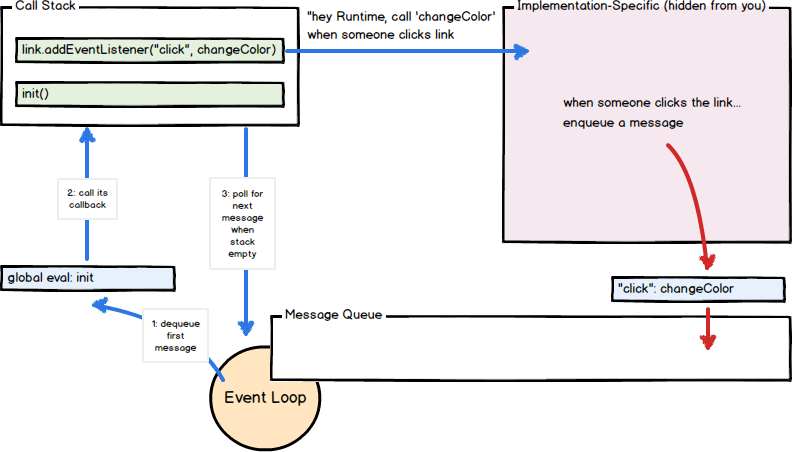
The call stack is a LIFO queue (Last In, First Out).

The event loop continuously checks the **call stack** to see if there’s any function that needs to run. While doing so, it adds any function call it finds to the call stack and executes each one in order.

**Web API**

Browsers have defined API’s which developers can be used to make complex processes such as to get location of visitor, GeoLocation is defined. A list of APIs are defined in the link which you can find in references *(1)*.

**The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there’s nothing in there, it goes to pick up things in the message queue.**



**Note:** In general, in most browsers there is an event loop for every browser tab, to make every process isolated and avoid a web page with infinite loops or heavy processing to block your entire browser.

Web API, Message / Callback Queue and Event Loop mechanisms are part of browsers.

**Example:**

const bar = () => console.log('bar')

const baz = () => console.log('baz')

const foo = () => {

console.log('foo')

setTimeout(bar, 0)

baz()

}

foo()

This code prints, maybe surprisingly:

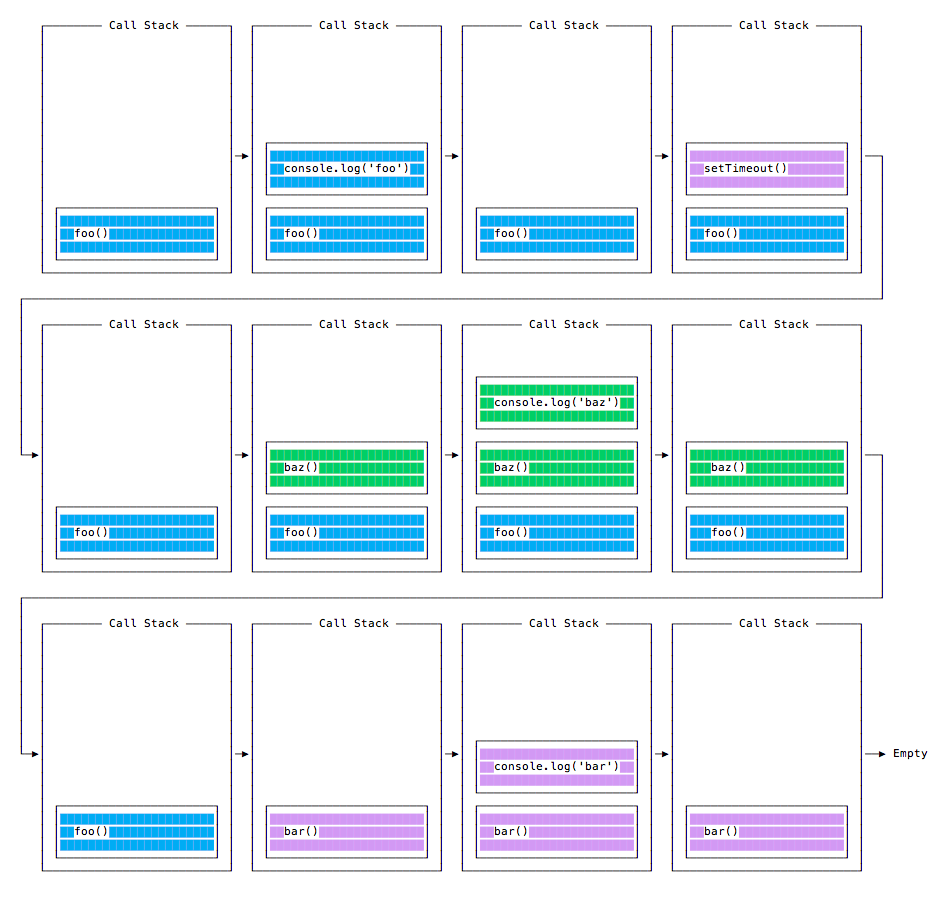
foo

baz

bar

When this code runs, first foo() is called. Inside foo() we first call setTimeout, passing bar as an argument, and we instruct it to run immediately as fast as it can, passing 0 as the timer. Then we call baz().

At this point the call stack looks like this:



Memory Management in JavaScript

The primary goal of memory management is to offer the system dynamically allocated memory when requested for and later free that memory containing objects that are no longer in use.

* Whenever you define a variable, constant, object, etc in your javascript program, you need some place to store it. This place is nothing but the memory heap.
* When the statement var a = 10 is encountered, a location in the memory is assigned to store the value of a.

## Garbage Collection

The JavaScript Engine’s Garbage collector basically looks out for unreachable objects which are removed from the memory. There are two garbage collection algorithms that I would like to explain which are as follows:

* Reference-counting garbage collection
* Mark-and-sweep algorithm

## Reference-counting garbage collection

This is a naïve garbage collection algorithm. 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.

var obj1 = { property1: { subproperty1: 20 }  
};

Let’s create an object as shown in the above example to understand this algorithm. Here obj1 has an object in which its property1 holds further one object. As the obj1 has the reference to the object, nothing is eligible for garbage collection.

var obj2 = obj1;obj1 = "some random text"

Now, obj2 also has the reference to the same object that was referred by obj1 but later obj1 was updated with "some random text" which lead to obj2 having the unique reference to that object.

var obj\_property1 = obj2.property1;

Now obj\_property1 refers to obj2.property1 which also holds an object. Therefore that object has two references which are as follows:

1. As a property of obj2
2. In the variable obj\_property1

obj2 = "some random text"

obj2 has been unreferenced by updating "some random text". Therefore it might seem that the object it held before has no references to it and can be garbage collected. But that might be wrong to say as obj\_property1has the reference of obj2.property1. Therefore it won’t be garbage collected.

obj\_property1 = null;

Now the object which was originally in obj1 has no references left as we removed the reference from obj\_property1. So now it can be garbage collected.

## Where does this algorithm fail?

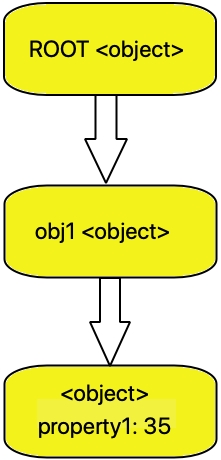
function example() { var obj1 = { property1 : { subproperty1: 20 } }; var obj2 = obj1.property1; obj2.property1 = obj1; return 'some random text'}example();

Here the reference counting algorithm does not remove obj1 and obj2 from the memory after the function call, since both the objects are referenced by each other.

## Mark-and-Sweep Algorithm

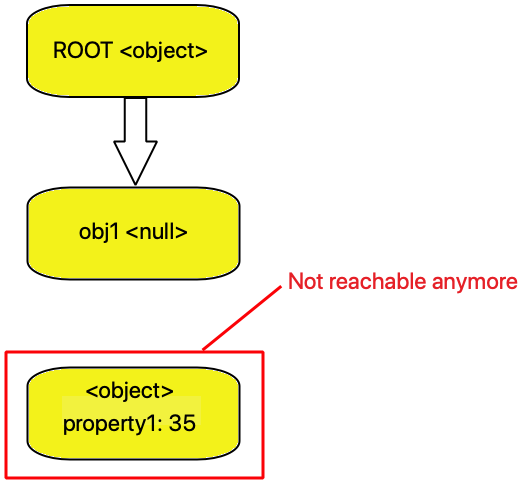
This algorithm looks out for objects which are unreachable from the root which is the JavaScript’s global object. This algorithm overcomes the limitations of Reference-counting algorithm. As an object with no references would be unreachable but not vice-versa.

var obj1 = { property1: 35}



As shown above, we can see how the created object obj1 becomes reachable from the ROOT.

obj1 = null



Now when we set the value of obj1to null the object is no more reachable from the ROOT and hence it is garbage collected.

Memory leaks

Although the garbage collection is effective, the developers should not be under the impression that they need not worry about memory management. Managing memory is a complex procedure and which piece of memory is not required cannot be decided by an algorithm.

Memory leaks are parts of memory that the application needed and used in the past and it is not needed anymore but its storage is yet not returned to the memory pool.

Following are some of the common mistakes that cause this memory leaks in your application.

1. **Global variables:** If you keep creating global variables, they will stick around throughout the execution of the program even if they are not needed. If these variables are deeply nested objects, a lot of memory is wasted.

var a = { ... }  
var b = { ... }  
function hello() {  
 c = a; // this is the global variable that you aren't aware of.  
}

If you try to access a variable which was not declared previously, you will create a variable in the global scope. In the above example, ‘c’ is that variable/object that you didn’t implicitly create using the ‘var’ keyword.

1. **Event Listeners:** This may happen when you create a lot of event listeners to make your website interactive or maybe just for those flashy animations and forget to remove them when the user moves to some other page in your single page application. Now when the user moves back and forth between these pages, these listeners keep adding up.

var element = document.getElementById('button');  
element.addEventListener('click', onClick)

1. **Intervals and Timeouts:**When referencing objects inside these closures, the garbage collector will never clear the objects until the closure themselves are cleared.

setInterval(() => {  
 // reference objects  
}  
// now forget to clear the interval.  
// you just created a memory leak!

1. **Removed DOM elements:** This one is similar to the global variable memory leak and very common. DOM elements exist in the Object Graph memory and the DOM tree. This scenario is better explained by an example.

var terminator = document.getElementById('terminate');  
var badElem = document.getElementById('toDelete');  
terminator.addEventListener('click', function() {memory  
 badElem.remove();  
});

After you click the button with id = ‘terminate’ , toDelete is removed from the DOM. But since it’s still referenced within the listener, the allocated memory for the object is still used.

var terminator = document.getElementById('terminate');  
terminator.addEventListener('click', function() {  
 var badElem = document.getElementById('toDelete');  
 badElem.remove();  
});

Now, the badElem variable is a local variable and when the remove operation completes, the memory can be reclaimed by the garbage collector.

10 Tips for Effective Web Design

1. Simplicity (in web design less is more, don’t clutter / add too much text on your web page)
2. Consistency (Stay consistent with font, font-family, size of heading, and style of images)
3. Readability (contrast between image and background, spacing, line-height)
4. Responsiveness (mobile compatibility, use media queries, bootstrap)
5. Simple Navigation (good to have the nav bar at the top of the page and it should be sticky)
6. Having a purpose (for the person going to the site, convey your services and product and a way contact you)
7. Call to action (Adding contact form, Sign up form, registration form, add to cart)
8. Color pallet (have branding color, there is good contract, light-blue - darker blur - white just as an example of color pallet)
9. Load speed (optimize the website, big culprit of this is images, like if there is 5000 px image and you don’t compress it then it is going to take ling time to load)
10. Minimal advertising (place adds strategically and try to make it look elegant)

## Stack overflow question about closure concept

function getObj()

{

var objAddress =

{

address: "Client Address",

getAddress: function() {

return this.address;

},

setAddress: function(newAddress)

{

this.address = newAddress;

}

};

var objClient =

{

name: "Client name",

getAddress: function()

{

return objAddress.getAddress();

},

setAddress: function(newAddress) {

objAddress.setAddress(newAddress);

}

};

return objClient;

}

gObj = getObj();

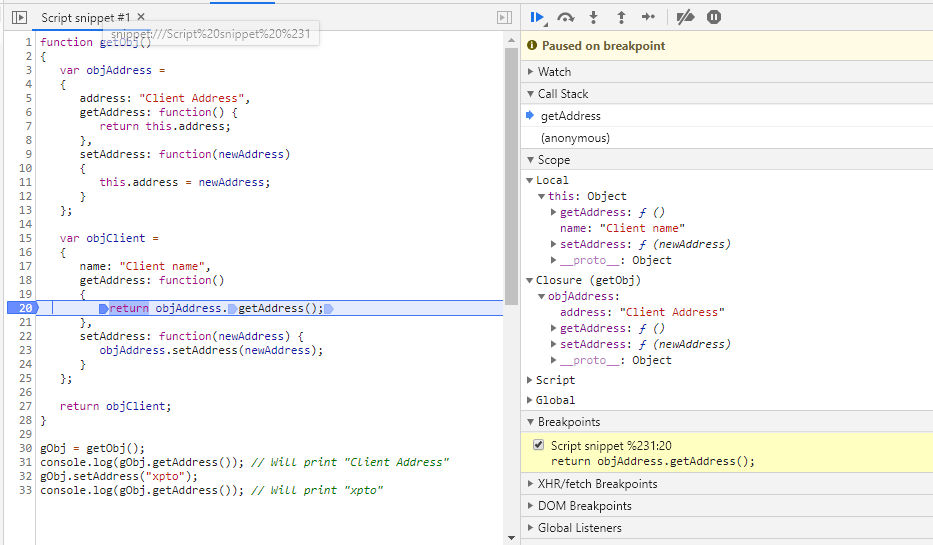
console.log(gObj.getAddress()); // Will print "Client Address"

gObj.setAddress("xpto");

console.log(gObj.getAddress()); // Will print "xpto"

 A programmer thought it would not work since getAddress() calls another method of an object that should not exist after leaving the function. But, as this is working,

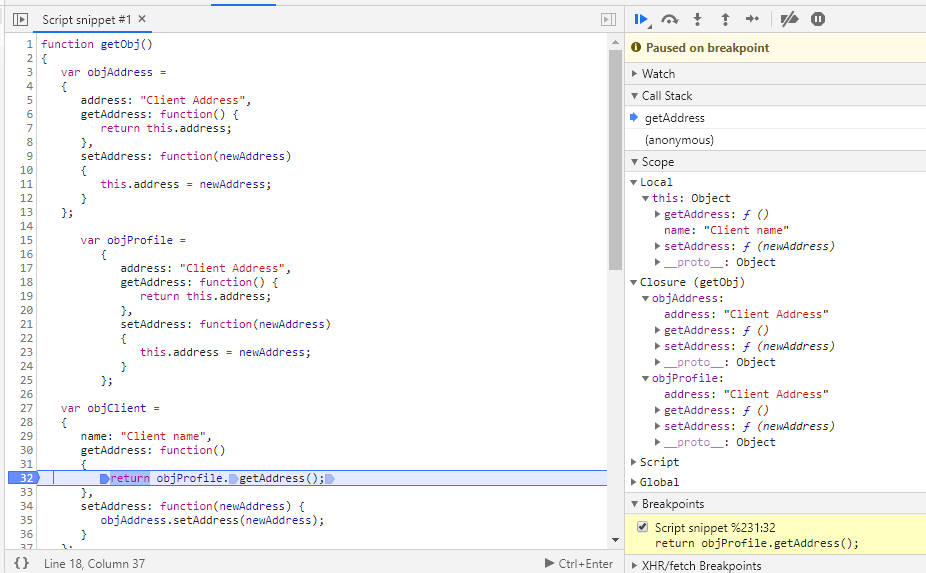
Chrome – Developer Tools



Set the breakpoint at line number 20 and run the code snippet.

On the right side, you will be able to see the local variables and closures coming within the scope of line in execution.

If a new object names objProfile is added to getObj function then that object will also become part of closure.



*Comment from stack overflow:*

When you create function, which uses local variable, function "remembers" this local variable. All needed local variables are stored in special object called closure. You cannot directly access it, but function can. Chrome Developer Tools in javascript debug shows closure object:

Under the hood: variables lifecycle

When the engine works with variables, their lifecycle consists of the following phases:

1. **Declaration phase** is registering a variable in the scope.
2. **Initialization phase** is allocating memory and creating a binding for the variable in the scope. At this step the variable is automatically initialized with undefined.
3. **Assignment phase** is assigning a value to the initialized variable.

## var variables lifecycle

The variable passes the declaration phase and right away the initialization phase at the beginning of the scope, before any statements are executed

Strictly hoisting consists in the idea that a variable is declared and initialized at the beginning of the function scope.

## Function declaration lifecycle

The declaration, initialization and assignment phases happen at once at the beginning of the enclosing function scope (only one step).

## let variables lifecycle

let lifecycle however decouples declaration and initialization phases.

Now let’s study a scenario when the interpreter enters a block scope that contains a let variable statement. Immediately the variable passes the *declaration phase*, registering its name in the scope (step 1).  
Then interpreter continues parsing the block statements line by line.

If you try to access variable at this stage, JavaScript will throw ReferenceError: variable is not defined. It happens because the variable state is *uninitialized*.  
variable is in the *temporal dead zone*.