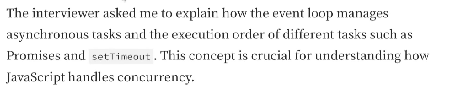
JAVASCRIPT INTERVIEW QUESTIONS

1. **Event Loops**



**Answer**:

So, JavaScript is a single-threaded synchronous language. Now, it has a single call-stack where the sequence is run line by line.   
This call-stack is inside JS Engine. The JS Engine is inside a browser.

Now, if we want to run a code after 5 seconds in JS. Can we, do it? Answer is NO. Because it is a synchronous language, and it does not have a timer functionality of its own.   
How can we do it? With the help of the browser. Browser has a lot of functionalities. And we can connect with those functionalities with the help of **Web APIs.**

Web APIs include, setTimeout(), DOM APIs(document.getEl…), fetch(), localStorage, even the console.log() isn’t part of JS.

We have a keyword in browser called **window,** through this keyword we can access all the Web APIs. For example, window.setTimeout(), window.localStorage() etc.

For example:

**console.log(“Start”);**

**setTimeout(function cb() {**

**console.log(“callback”);**

**}, 5000);**

**console.log(“End”);**

The code will run line-by-line (A Global Execution Context (GEC) is created inside the callstack, which helps the callstack run line-by-line). It calls web api console. It will push first console.log into call stack. Then, it moves to setTimeout(), it has some delay. Then, the code runs to next line which is the second console.log.   
So the output will be:   
Start

End

Callback

Even if the timer showed 0 ms in it, it would have first run the console statements then the timer as it takes time to call the Web APIs then run the code. So, there is a callback queue, the setTimeout() is pushed in the callback queue, and then the event loop comes in the picture. Event loop acts as a gatekeeper and checks if there is something in the callback queue. If there is something, it pushes inside the call stack. So, it pushes the cb function inside callback queue… and it will execute the cb func. Line by line. Here it only had one line, which was a console.log().

Example 2:

**console.log(“Start”);**

**document.getElementById(“btn”)**

**.addEventListener(“click”, function cb() {**

**console.log(“callback”);**

**});**

**console.log(“End”);**

Output:

Start

End

Callback

Whenever we come across DOM APIs, here when it encounters addEventListener, inside it, it has a callback function which gets registered and attaches an event to it, which is a click event. Now, the second console.log() runs, so end gets printed.. When the user clicks on the button, the cb method sits inside the callback queue and wait for it turn to get executed.

Now, event loop has only one job, to continuously monitor the call stack and callback queue. It takes the already waiting cb method, and pushes it to call stack. So it gets executed.

A screenshot of a computer

Description automatically generated

Why do we need callback queue?

Because, if the user clicks on the button 7-8 times, so everytime a cb method will be pushed in cb queue (7-8 times).

A close-up of a line of squares

Description automatically generated

It takes according to FIFO and pushes in the stack.

Example 3:

**console.log(“start”);**

**setTimeout(function cbT() {**

**console.log(“CB SetTimeout”);**

**}, 5000);**

**fetch(**[**https://api.netflix.com).then(function**](https://api.netflix.com).then(function) **cbF() {**

**console.log(“CB Netflix”);**

**});**

**console.log(“End”);**

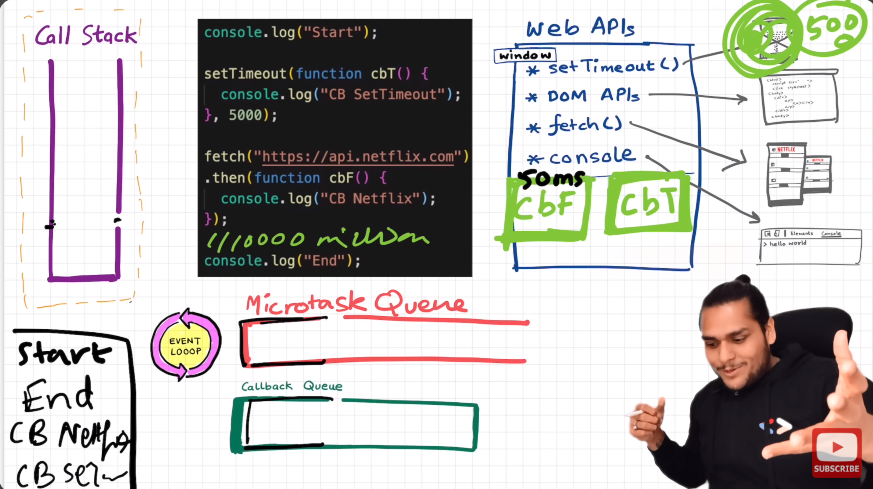
GEC is created, Start is printed. setTimeout will register a cb function. Timer of 5000ms also started. JS Engine moves to next line. Encounter fetch, it also registers cb function into WEB APIs environment (cbF). cbT is waiting for the timer to get finished, while the cbF function is waiting for data to be returned from the mentioned API. Now, we also have something as a **microtask queue**.. But this queue has higher priority. cbF will go to the microtask queue. Job of event loop is to continuously monitor if the callstack is empty or not. End will be printed there first. Event loop is empty now, cbF will be pushed first as it has the higher priority. So prints CB Netflix. Now cbT is pushed, which executes CB SetTimeout.

Start

End

CB Netflix

CB SetTimeout



What can come in the Microtask queue?

All the cb functions which comes through promises will go inside this queue. And, mutation observers go inside this queue.

Starvation of the callback queue?

When one microtask creates another microtask and it goes on.. So the task in the cb queue never gets a chance to execute. This is called Starvation of the callback queue.

1. **Promises and Async/Await**

|- Different types of Promise APIs

|- Promises Chaining

|- Callback Hell (First view this)

A close up of a text

Description automatically generated

**CALLBACK HELL:**

**For example:**

**const cart = [“shoes”, “pants”, “kurta”];**

**api.createOrder (cart, function () {**

**api.proceedToPayment(function () {**

**api.showOrderSummary(**

**function() {**

**api.updateWallet()**

**}**

**)**

**}**

}

Here, we can see that after the order is created, we have another callback function inside it which will be called after the order is created. After the payment is done, another cb function is called which is show order summary. So here, if we notice we are stuck in a callback hell, where there is one callback inside another callback and so on….

This is also called **Pyramid Of Doom.**

Another problem we face while using callbacks is **Inversion of Control.**

Losing the control of code while using callbacks. We have given to control of a function to another function and we don’t know if the first function will call the cb function correctly.

**PROMISES:**

From the last example,

**const cart = [“shoes”, “pants”, “kurta”];**

**api.createOrder (cart, function () {**

**api.proceedToPayment(orderId);**

**});**

**const promise = createOrder(cart);**

Promise is an empty object with some data value. When the last line executes, it will return us a promise. Since, the createOrder is an async task (meaning, we don’t know how much time it will take to execute). So, it will fill this promise object automatically after an async time.

We will attach a callback function to this promise object.

**promise.then( function () {**

**api.proceedToPayment(orderId);**

**});**

Once we have data filled in promise object, this function will automatically be called. This type of practice is much much better than previous callback hell.

How?

In callback hell, we were **passing** a cb function and blindly trusting it will execute, which is risky. But in this case, we are **attaching not passing** a function to a promise object. Here, when the data is returned to a promise, then the function will be called. That means, this is in our control. Here the guarantee also is, when the data inside promise, it will 100% call the function and only once.

**For Example:**

**const API =** [**https://jjknfjewfnj.com**](https://jjknfjewfnj.com)**;**

**const user = fetch(API);**

**console.log(API);**

Here, fetch returns a PROMISE and is stored in the user variable. In chrome, when we try this out, if we run it now.. it will show us pending state. So promise has 3 states, **pending, fulfilled and rejected**. In chrome, as of we are talking now (Year 2024). The chrome browser will log it as pending, but if we expand it , it will show us fulfilled. Just another Chrome browser issue.

Promises are immutable (meaning, it will not change and we can blindly pass that data or play with it).

What is a Promise?

Promise is a placeholder, which will be filled later with a value from an async operation.

OR

Promise is an object representing the eventual completion or failure of an async operation. (Use this if possible)

**PROMISE CHAINING:**

**createOrder(cart).then(function (orderId) {**

**return proceedtoPayment(orderId)) //** Remember to return, so that the value passes on to next chain.

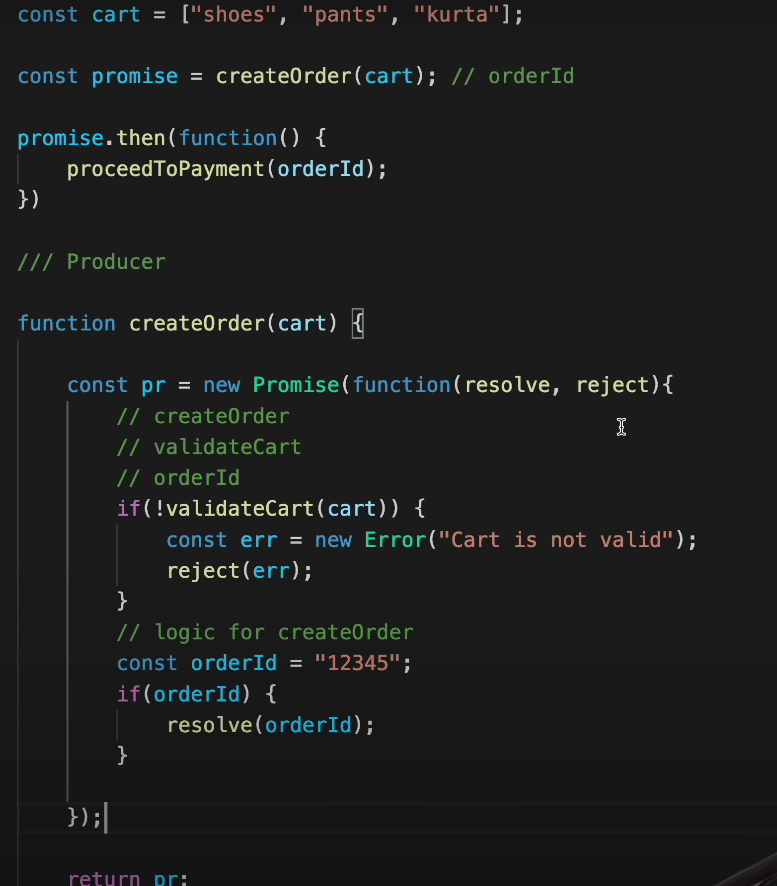
**}.then (function (paymentInfo) {**

**return showOrderSummary(paymentInfo))**

**}**

How to create a Promise?

**Same example of cart…**

****

Here, we write the logic of creating an order. We create a promise with **new Promise(resolve, reject)** keyword. And now we have written of the cart validation. If there is an error, we use **reject** keyword otherwise we use **resolve** keyword. So, when we call the createOrder API, a promise is created.

Handling errors with catch statements.

A screen shot of a computer code

Description automatically generated

This is promise chaining. Remember to return it so that the values can be passed down the chain.

A screen shot of a computer program

Description automatically generated

What if the cart was not validated and still, we wanted to proceed to payment?

Here, we just move the catch statement above in the chain. Now, the catch statement has the responsibility to check only which is on top of it. It will not check what is below catch statement.

A screen shot of a computer program

Description automatically generated

**PROMISE APIs:**

**TERMS:**

**A blackboard with writing on it

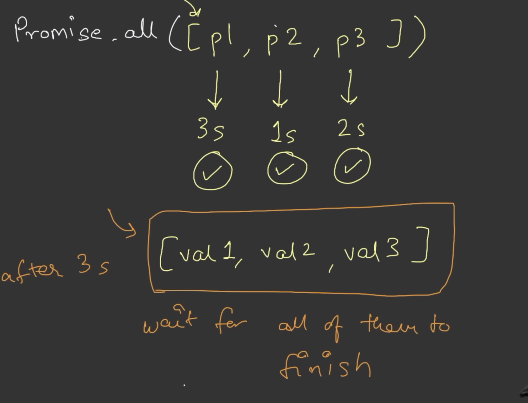
Description automatically generated**

**1. Promise.all()**

* If we had to make parallel API calls
* Suppose if we had 10 user id’s and we want an API call for user info and when we want to make the API call and get the result of 10 different users.
* Takes an array of promises
* It returns an array with the result of all of the promises.
* It will wait for all of them to finish before returning the array of output.

Here, it will return the output after 3 seconds if every promise is successful.

Why 3 seconds? As the total time here is 3 seconds and everything is running in parallel. So it will wait until everything is finished.



Now, what if one of them gets rejected?

As soon as one of them gets rejected**, Promise.all() will throw an error with the same error of the one which got rejected.**

So that means, here in the above example, every promise is running parallelly. So, after 1 second it will throw and error and not wait for others to finish.

What about p1 and p3? Will they get cancelled? No, they will not get cancelled but the end result of promise.all() will be a reject as one of them got rejected.

CODE**:**

A screenshot of a computer

Description automatically generated

After 1 second it failed.

Now, what if we wanted the successful promises even when one of them or few of them gets rejected. How do we do that?

We have,

2. **Promise.allSettled()** for it. Even when one of them gets rejected, it will wait for all promises to settle. That means, it will wait for 3 seconds, it will return the array of output.

It will display status of all 3, if gets fulfilled it will show that as well as the error of rejection.

CODE:

A screenshot of a computer screen

Description automatically generatedWaited for all the promises to settle then returned the result.

**3. Promise.race()**

Taking the above example, if p1 takes 3s, p2 takes 5s, and p3 takes 2s. It will return the value of the one that gets finished first. That means, val3 will get returned from p3 after 2 seconds.

What if the first settled promise was rejected?

Error will be thrown.

It’s just like a race. It will return the results(resolved or rejected0 according to the timeline of promises. Only the first **settled (means not in a pending state)**  promise (Either resolved or rejected).

Returns a single promise.

CODE:

A screenshot of a computer

Description automatically generated

Returned P2’s success.

4. **Promise.any()**

Here**, it will wait for first settled success.**

If the first gets rejected, it will wait for the success and not return anything. Returns a single promise. **Seeking first success**

What if everything fails?

Returned result will be an aggregate error. It will be an array of all the errors.

CODE:

A screenshot of a computer screen

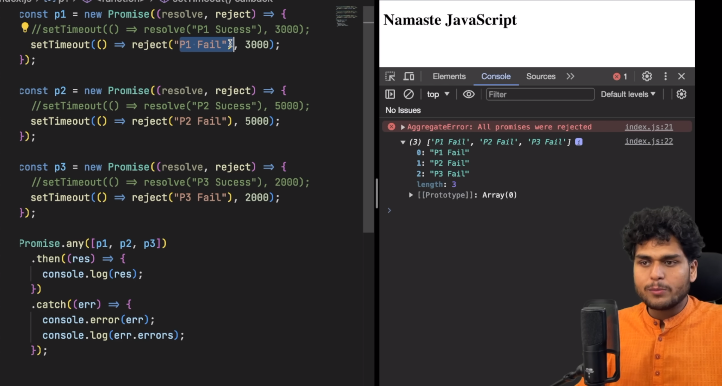
Description automatically generated

Returned first settled success [p1] after 3 seconds even when p2 got an error after 2 seconds.

A screenshot of a computer screen

Description automatically generated

Aggregate error came when everything failed.



Check the code here. We said that we will get an array of errors but we did not at first. Check the catch statement. That’s how you print the array of errors.

We had to do a **console.log()**

**ASYNC/AWAIT :**

What is async?

It is a keyword that is used before a function to make it asynchronous.

How is it different from a normal function?

* **Async func. always return a promise**

It can return directly a promise. Or, if we find out that it is returning, for example, a string. It will wrap that string inside a promise and return it.



It returned a promise.

A computer screen shot of a computer code

Description automatically generated

Here, when the promise is returned from the function, it is stored inside a value **dataPromise.** Now , when we do .then(), the result will log the actual value, which is **Namaste**.

A computer code with colorful text

Description automatically generated

Here, in the code we created another promise. And inside the async function, we called “p”.

What will happen here? Will it wrap a promise into another promise and return it? No, if the return value already a promise, it will be returned as it is.

**Async and await combo is used to handle promises.**

How did we used to handle promises before async/await?

A computer screen shot of a computer code

Description automatically generated

We just called a (res) and console.log() the value.



Here, we will use a keyword await to handle promises. The variable “val” will contain the value of the resolved promise.

A black background with purple lights

Description automatically generated

Same result.

What is await?

Await is a keyword that can be used only inside an async function.

What is the difference between async/await and normal promise handling?

A screen shot of a computer program

Description automatically generated

In the code, when we call the getData() function, what do you think is going to happen? What will be the output? Will console.log() be printed first or the resolved promise value?

The console.log() will be printed first. JSEngine will not wait for promise to be resolved and move on to the next line.

Output:

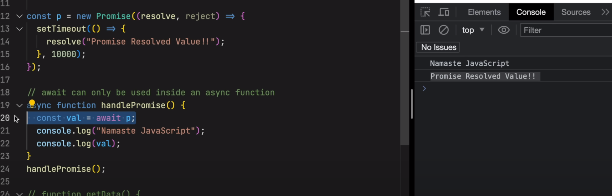
Namaste Javascript

Promise resolved value!!

The promise resolved value would be printed after 10 seconds. If we changed the setTimeout to run after 0ms. Both will be printed at the same time.

This is how the promises were handled before.

Now with async/await…



Here, the code will wait on line no.20 till the promise resolves and then print both the values together. JSEngine will wait for promise to get resolved. Will only go to next line once the promise is resolved.

NOTE: If we had a console.log() above the await keyword. That would have been executed first then, the engine would wait for promise to get resolved and then move on to the next line.

A screenshot of a computer

Description automatically generated

If we had await keyword 2 times. What will happen here? Will the program will wait for two times? Or will it run parallelly? It will run parallelly and print both of them on the same time. Will not wait for two times.

Now what about this code:

**const p1 = new Promise((resolve, reject) => {**

**setTimeout(() => {**

**resolve('Hello');**

**}, 10000)**

**})**

**const p2 = new Promise((resolve, reject) => {**

**setTimeout(() => {**

**resolve('Hello Again');**

**}, 5000)**

**})**

**async function handlePromise() {**

**console.log("Will I get executed first?");**

**const val = await p1;**

**console.log("Namaste!");**

**console.log(val);**

**const val2 = await p2;**

**console.log("Namaste!");**

**console.log(val2);**

**}**

**handlePromise();**

What will happen here?

After 10 seconds, all of it will be printed. The promise which had 5 seconds got resolved but still waited for 10 seconds.

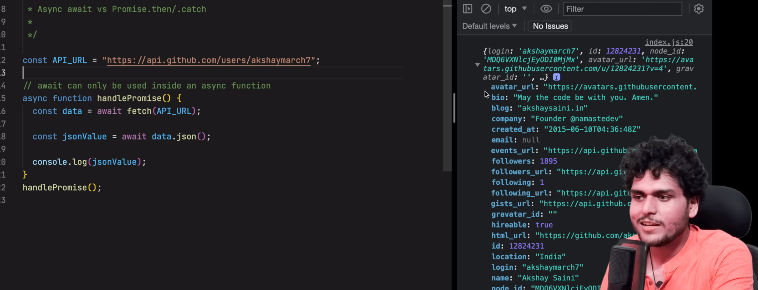
Now what about if we reversed the timeouts. If p1 had 5 seconds and p2 had 10 seconds. What will happen now? After 5 seconds the p1 resolved value will be printed and then the next promise will be resolved and printed. This is confusing and these are few of the interview questions that get asked around this. And ofcourse the console.log value above await keyword will be printed first before all of this.

How will this work behind the scenes?

* The **call stack** starts empty.
* When the function handlePromise() is called, it enters the call stack.
* Since JavaScript is **single-threaded**, it executes the code line by line.
* The first console.log (above the await keyword) is printed immediately.
* The next line contains await p1:
  + Encountering the await keyword suspends the execution of handlePromise().
  + The function moves out of the call stack and does not block the page.
* Once p1 resolves (after 5 seconds):
  + handlePromise() re-enters the call stack.
  + Execution resumes from where it was suspended.
  + It logs the values "Namaste" and "Hello".
* The next line contains await p2:
  + If p2 is unresolved, the execution is suspended again.
  + The call stack becomes free.
* Once p2 resolves:
  + handlePromise() re-enters the call stack.
  + Execution resumes, logging "Namaste!" and "Hello Again".

How Fetch Works?

So fetch is a promise, when this promise is resolved, it gives us a response object. This response object has a body, which as a readable stream. This response body is a readbale stream. If we have to convert this readbale stream to json, we have to use .json(). This json() is again a promise. When this promise is resolved, it gives us the json value.



How will we handle errors in async await?

We will use try{}catch{} method.

A screenshot of a computer

Description automatically generated

In conclusion, ASYNC/AWAIT IS JUST A SYNTACTICAL SUGAR FOR HANDLING PROMISES. BOTH OF THESE METHODS TO HANDLE PROMISES ARE GOOD BUT IT HAS FEW DIFFERENCES WHICH WE STUDIED. IT’S JUST A NEW WAY OF WRITING CODE. HERE WE DON’T HAVE TO DO PROMISE CHAINING WHILE WE USE ASYNC / AWAIT.

IT'S A PERSONAL CHOICE.

1. **Hoisting**

A close up of a text

Description automatically generated

Here, if we wrote a code:

**var x = 7;**

**function getName() {**

**console.log(“Namaste JavaScript!”);**

**}**

**getName();**

**console.log(x);**

Output:

Namaste Javascript

7

This works fine. But what if we did something like this:

**getName(); //**it will read this function even If we move it above the declaration

**console.log(x); //** it will give us undefined here

**var x = 7;**

**function getName() {**

**console.log(“Namaste JavaScript!”);**

**}**

**Output:**

Namaste JavaScript

Undefined

Now if we removed the initialisation of x and then ran the program, it would have given us an error.

This is called Hoisting. So, Hoisting is a phenomenon in JavaScript by which we can access the variables and functions even before we initialised it. We can access it without any error.

So, from where did this undefined keyword comes from?

This happens because even before the start of the execution of a program, JS skims through the program and it will allocate memory for these variables. So, it will store a special keyword undefined over here.

What happens in case of functions?

The whole code is put in. We will get an actual copy of a function.

The difference between not defined and undefined.

**undefined** is used when a variable is declared but not initialised.

**not defined** occurs when you try to access a variable that hasn’t been declared at all.

If we created getName as an arrow function:

**A screenshot of a computer

Description automatically generated**

This error will occur because here using arrow function, it will behave like a variable.

A person in a black shirt

Description automatically generated

Even if we create a function with this syntax, it will behave just like a variable and during the memory allocation process, it will be allocated undefined.

**NOTE: STUDY MORE ABOUT THIS TOPIC REGARDING VAR, LET, CONST Etc.**

1. **Closures**

|- Output questions

A close-up of a white background

Description automatically generated

Lexical Scoping:

**function x() {**

**var a = 7;**

**function y() {**

**console.log(a);**

**}**

**y();**

**}**

**x();**

**Output:**

**7**

Why 7? This is called lexical scoping.

Lexical scoping means that when you write a function inside another function, the inner function can access variables from its own scope, plus variables from its parent's scope. Think of it like a child who has their own toys (local variables) but can also use their parent's stuff (parent scope variables).

* Inner functions can look outside their scope to find variables
* Outer functions cannot look inside inner functions
* This "scope chain" always goes from inside to outside, never the other way

What is Closure?

A closure is formed when a function "remembers" and can access variables from its outer scope even after the outer function has finished executing. Think of it like a backpack that a function carries around, containing all the variables it had access to when it was created.

**Code:**

**function createCounter() {**

**let count = 0; // Private variable**

**return function() {**

**count++;**

**console.log(count);**

**return count;**

**}**

**}**

**const counter = createCounter();**

**counter(); // 1**

**counter(); // 2**

**counter(); // 3**

**const counter2 = createCounter();**

**counter2(); // 1**

**counter2(); // 2**

Let me explain closures with a simple counter example. A closure is formed when a function retains access to variables from its outer scope even after the outer function has finished executing.

In this code:

1. createCounter creates a variable count and returns a function
2. The returned function has access to count even after createCounter finishes running
3. Each time we call counter(), it "remembers" and updates the same count variable
4. If we create a new counter (counter2), it gets its own separate count variable

**INTERVIEW QUESTION:**

**function outer() {**

**for(var i = 1 ; i <= 5; i++) {**

**setTimeout(function () {**

**console.log(i)**

**}, i \* 1000);**

**}**

**console.log("Namaste Javascript!");**

**}**

**outer();**If I run this code, the output is going to be 6 (5 times) after a second each. But if I replace **var i = 1 to let i = 1.** The output runs perfectly , which is 1 2 3 4 5.

Why?

With var:

What happens:

* var creates ONE single i that's shared by all setTimeout callbacks
* The loop finishes running (i becomes 6) before any setTimeout executes
* When the timeouts finally run, they all reference the same i, which is now 6
* So you get: 6, 6, 6, 6, 6

With let:

What happens:

* let creates a NEW i for each loop iteration
* Each setTimeout callback captures its own copy of i
* When the timeouts run, each has its own preserved value of i
* So you get: 1, 2, 3, 4, 5

Now, what if the interviewer asks to run it through var only, we cannot use let for this. Produce the same output with var.

Answer:

**function outer() {**

**for(var i = 1 ; i <= 5; i++) {**

**function close(i) {**

**setTimeout(function () {**

**console.log(i)**

**}, i \* 1000);**

**}**

**close(i)**

**}**

**console.log("Namaste Javascript!");**

**}**

**outer();**

Here, using close function, we created a new copy of i, whenever we called setTimeout.

Here's what's happening:

1. Each time the loop runs, it calls close(i) with the current value of i
2. The close function creates a new scope each time it's called
3. Each timeout function forms a closure with its own copy of i from its close function's parameters
4. So each timeout "remembers" its own unique value of i

Think of it like this:

* First loop: close(1) creates a scope with i = 1
* Second loop: close(2) creates a scope with i = 2
* And so on...

It's similar to giving each setTimeout its own private mailbox (scope) with its own number inside, rather than having them all share the same mailbox like in the original var example.

This works because:

1. Function parameters create a new scope
2. Each call to close() gets its own parameter i
3. The setTimeout closure captures this scoped parameter

This is essentially doing manually what let does automatically in the loop!

1. Prototypes and Inheritance

|- Generally theory questions

A close-up of a white background

Description automatically generated

1. **This Keyword**

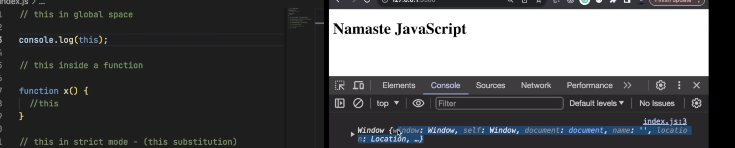
|- Always remember if using this keyword with normal fn. Or arrow fn. can be different

A close-up of a white background

Description automatically generated

**“this" in global space:**

Now, if we do a console.log(this) inside global space. We get an output of a **window** object.



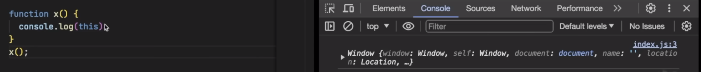
Why the value is window object?

**this** in global space will always have the value of global object, which is window object. In browser, the global object is **window.**

**The answer for us here depends on the global object. It can be window or something else. Depends upon where we are running our code.**

**“this” inside a function:**

// outputs to window object again.



What is the difference? Is it same?

No, it is not same, it is different. So when we run the function in a normal javascript environment. It is called as “unstrict mode”. In this , the value will come out to be window object inside a function.

But if we use strict mode in javascript. The output comes out to be **undefined.**

A screenshot of a computer

Description automatically generated

Why this happens?

It is because of “this substitution”. If the value of **this** is undefined or null. **this**  will be replaced with global object (Non-strict mode).

So how to answer a question like: **What is the value of this inside a function?**

The value of **this** will be undefined if we use **strict mode** but because of this substitution, we have the value as globalObject in non-strict mode. Inside non-strict mode, we have concept called as **this substitution** because of which we get globalObject.

If we use strict mode, and we call the function as:

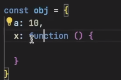


The output comes out to be a window object now. // window.x();

It depends on how the function is called. If we call it like x();// without reference of object. We will get an output as undefined. If we called it with reference to the object // window.x(); We will get output as the window object. (This is all run in strict mode).

**Q. What is the difference between a function and a method in Javascript?**

When we create a function inside an object. It is called as method.



Here, x is a method.

A computer screen shot of a program code

Description automatically generated

For this piece of code. The output of the value of **this** is:



The object itself.

If we did **this.a(), output // 10**

**Call/apply/bind methods: (Watch the video for call/apply/bind)**

**A screen shot of a computer program

Description automatically generated**

**Call is basically used for sharing the content of another object.**

Here's what's happening:

1. When you use student.printName()
   * this points to whatever is before the dot (.)
   * So this refers to student object
   * Therefore this.name prints "John"
2. When you use student.printName.call(student2)
   * call method lets you borrow a function and use it with a different object
   * It says "hey, use this function, but pretend this is student2"
   * So now this refers to student2
   * Therefore this.name prints "Kalpit"

Think of call like borrowing a friend's method:

* student2 is borrowing the printName function from student
* call lets you say "use this function, but with my data instead"

**There's also apply and bind which work similarly:**

**CODE:**

**student.printName.apply(student2); // Same as call**

**const newFn = student.printName.bind(student2); // Creates new function**

**newFn(); // Prints: "Kalpit"**

Behaviour of **this keyword** inside arrow functions:

Arrow functions don’t provide their own **this binding.**

A black rectangular object with white text

Description automatically generated

Here, in this code, the value of **this** will not be according to the function. It will be according to its “enclosing lexical environment”. So here, the value is again a window object.

Now,

A computer screen shot of a code

Description automatically generated

For this piece of code. What will be the output?

Here the output will be:

A computer screen with white text

Description automatically generated

The object itself. Why?

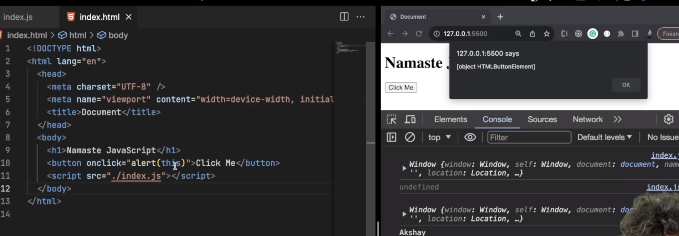
Check its enclosing lexical environment. The enclosing lexical environment is the function in method x(). That is why, it is printing the object itself.

A screen shot of a computer program

Description automatically generated

It will behave exactly like this code.

Behaviour of **this keyword** inside DOM:



In this piece of code, the value of **this** will be the value of button element itself.

So if the question is: **What is the value of this inside DOM?**

**this** inside DOM elements => reference to HTMLElement.

1. Flattening Objects

A white background with black text

Description automatically generated