**Q1. What is javascript and how it’s work?**

JavaScript is a client-side scripting language and one of the most efficient, commonly used scripting languages. The term .client-side scripting language means that it runs at the client-side( or on the client machine) inside the web-browsers, but one important thing to remember is that client's web-browser also needs to support the JavaScript or it must be JavaScript enabled.

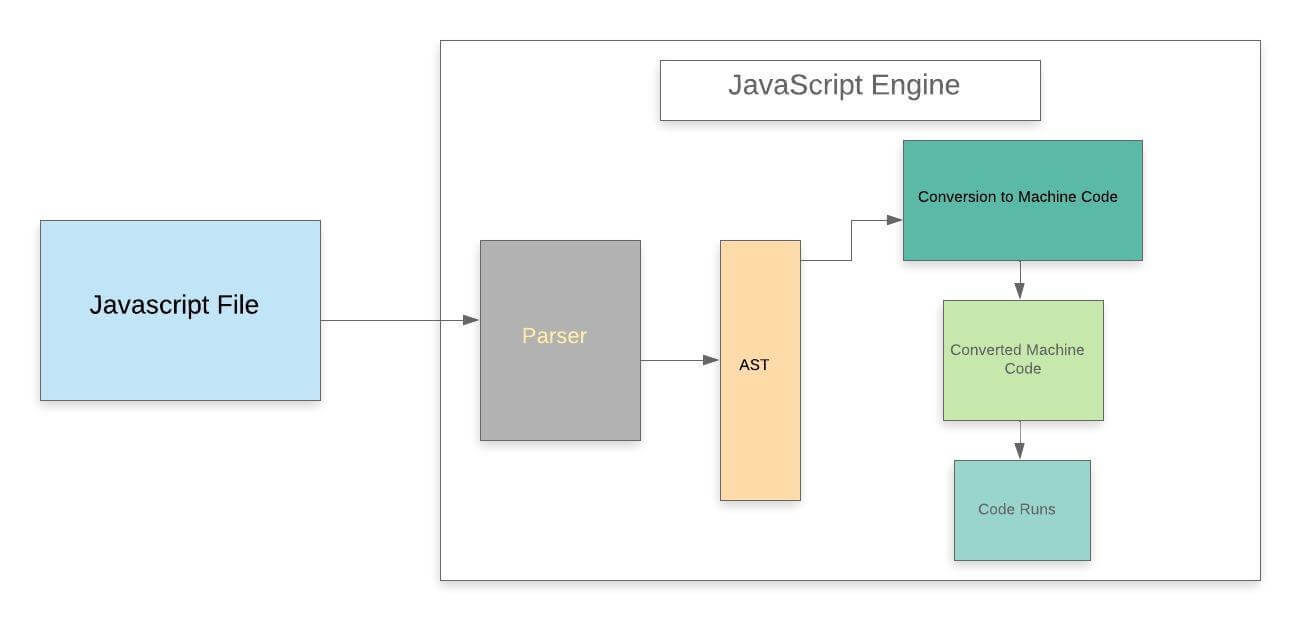
**Some other web-browsers with their JavaScript engines**

|  |  |  |
| --- | --- | --- |
| S.No. | **Web Browser** | **JavaScript engines** |
| 1. | Edge | Chakra |
| 2. | Safari | JavaScript Core |
| 3. | Firefox | Spidermonkey |

4. Google Chrome V8

**Working Of Javascript**

We can understand how a typical JavaScript engine works with help of a diagram:



**Step 1: Parser**

This is the first stage of the engine, every time we run a JavaScript program, our code is first received by the "parser" inside the JS engine. The parser's job is to check the JavaScript code for syntactic errors in line by line manner because JavaScript is an interpretive scripting language, so whenever an error is detected by the parser, it throws a kind of error and stops execution of the code.

In short, we can say that it parses JavaScript code.

**Step 2: AST**

Once the parser checks all JavaScript codes and gets satisfied that there are no mistakes/errors in the code, it creates the data structure called AST (it stands for Abstract Syntax Tree).

We can easily understand what is AST with help of following example.

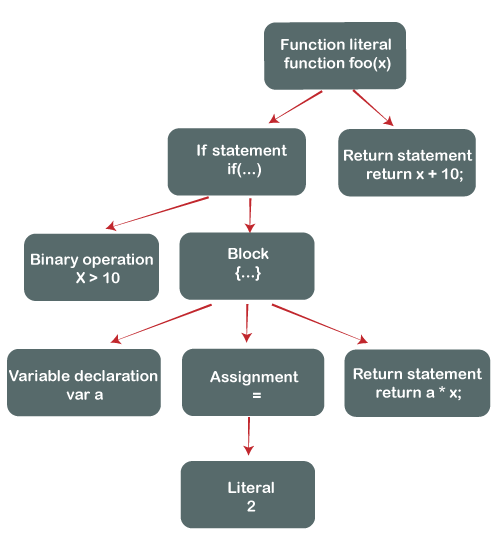
**Example**

Let's suppose we have a JavaScript program as given below:

**Program**

1. function fun(x) {
2. if (x **>** 15) {
3. var a = 4;
4. return a \* x;
5. }
7. return x + 10;
8. }

Once the parser checks the above JavaScript code, it will create a data structure called **AST** as we have already discussed above. The created AST (Abstract Syntax Tree) looks like the given image.



Note: It is not the exact abstract syntax tree, but it is the pictorial representation of the Abstract Syntax Tree.

**Step 3: Conversion to Machine code**

Once the Abstract Syntax Tree is created by the parser, the JavaScript engine converts the JavaScript code into the machine code (or in the language that machine can understand).

**Step 4: Machine code**

When the program written in the JavaScript gets converted in the machine language (or in byte code), the converted code is sent to the system for execution, and finally, that byte code run by the system/engine just like we observe in our first example.

**Q2. What is Execution Context?**

Execution context is the concept for describing the internal working of a js code.

It is an environment in which our code is executed and is evaluated.

It is the execution context that decides which code section has access to the functions, variables, and objects used in the code.

During the execution context, the specific code gets parsed line by line then the variables and functions are stored in the memory.

**Type Of Execution Context**

* Global Execution Context/GEC
* Functional Execution Context/FEC

**Global Execution Context/GEC**

JavaScript engine creates a GEC before it starts to execute any code.

GEC performs the two following task:

* Firstly, it creates a global object where it is for Node.js and Window object for the browsers.
* Secondly, reference the Windows object to 'this' keyword.
* Create a memory heap in order to store variables and function references.
* Then it stores all the functions declarations in the memory heap area and the variables in the GEC with initial values as 'undefined'.

Note: With the above introduction, one should understand that the Global Execution Context is only one in every code because the JS engine is single-threaded, and thus, only one global environment is possible for executing the JavaScript code.

**Functional Execution Context/FEC**

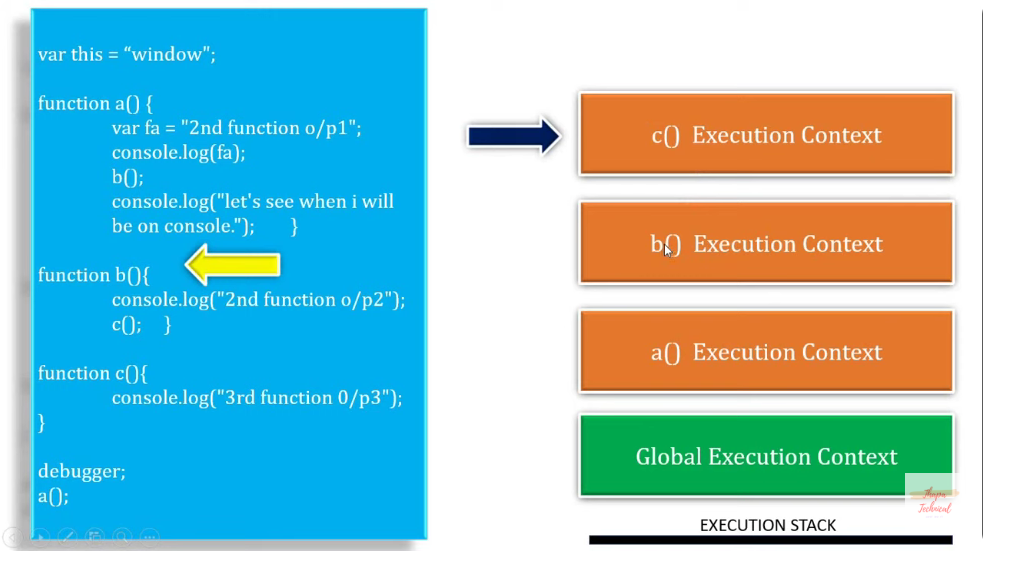
FEC or Functional Execution Code is that type of context which is created by the JavaScript engine when any function call is found.

A new execution context gets created every time when a function is executed.

FEC can access the entire code of the GEC, but it is not possible for GEC to access all the code of the FEC.

**Execution Stack**

It is also known as “calling stack” is a stack with a LIFO(Last In First Out), which is used to store all the execution context created during the code execution.

****

**Explanation**

Initially GEC will create before execute the code.

When the code executed it store the execution context in LIFO format and create a FEC.

For every function call new FEC create.

**Q3. What are the Phase Of Execution Context?**

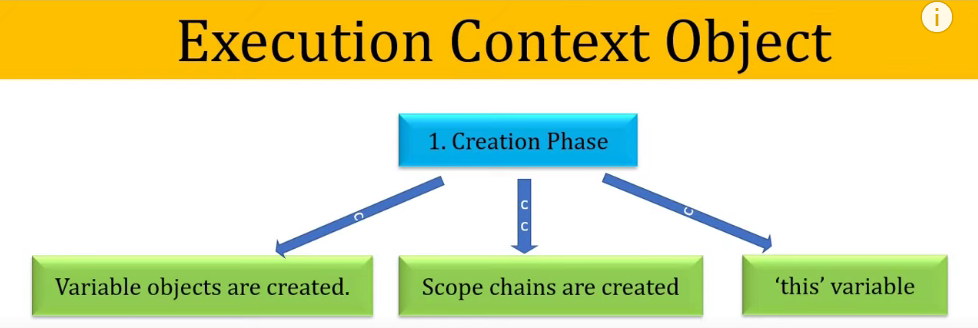
There are two phase of Execution Context

1. Creation Phase
2. Execution Phase

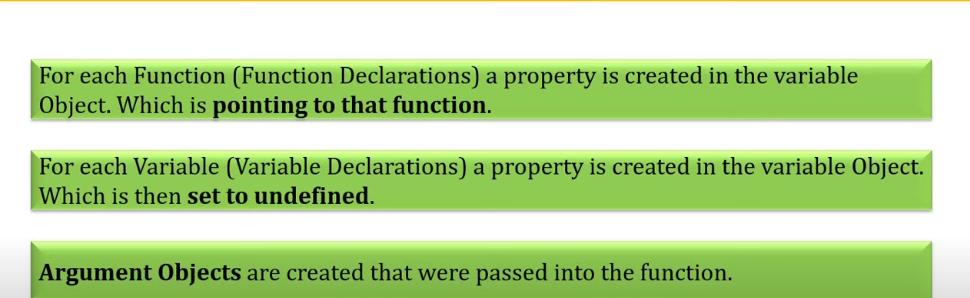
**Creation Phase**

It also have 3 part

1. Variable Object
2. Scope chain are created
3. `this` variable



* **Variable Object**

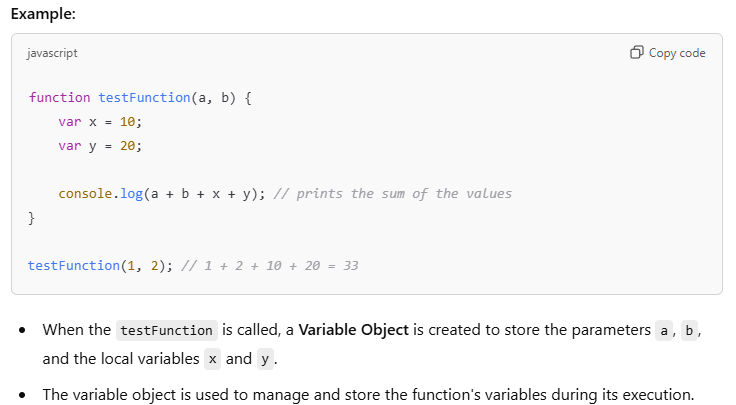
****

**OR**

In JavaScript, a "Variable Object" is part of the execution context, which holds all the variables, functions, and parameters in the context of function execution.

Every time a function is invoked, a **Variable Object** is created, and it stores:

* Function parameters
* Local variables
* Function declarations

****

**Q4 What is Hosting in javaScript?**

In [JavaScript](https://www.javatpoint.com/javascript-tutorial), Hoisting is a kind of default behaviour in which all the declarations either variable declaration or function declaration are moved at the top of their scope just before executing the program's code.

This happens during the 1st phase (memory creation phase) of the Execution Context.

**Note:** JavaScript only hoists declarations, not initializations.

Example

function sum(a,b){

    add = a+b;

    console.log(add);

}

sum(5,10);

In the above code in creation phase it moves the function at the top of the scope and gives the output

// Move at the top of scope

sum(5,10);

function sum(a,b){

    add = a+b;

    console.log(add);

}

**Note: It’s not work when we create a arrow function, function expression.**

sum(5,10);

const sum = (a,b)=>{

    add = a+b;

    console.log(add);

}

**It gives error because it send the variable at the top of the scope and at that time it assign undefine to variable.**

**Exercise On Hosting**

**Find the output**

****

**Output**

I am a undefine.

I am a web Developer.

I am a undefine.

I am a Full Stack

**Explanation**

When the first console run it get the “job” which is not define so in Creation Phase It moves to top and initialized with “undefined”.

For second console the job is already define so it console “Web Developer”

After that it moves to function call.

Same when the first console run inside the function the “job” is not define so in Creation Phase it moves to top of the scope and set “undefined”.

So, It console “I am undefined”.

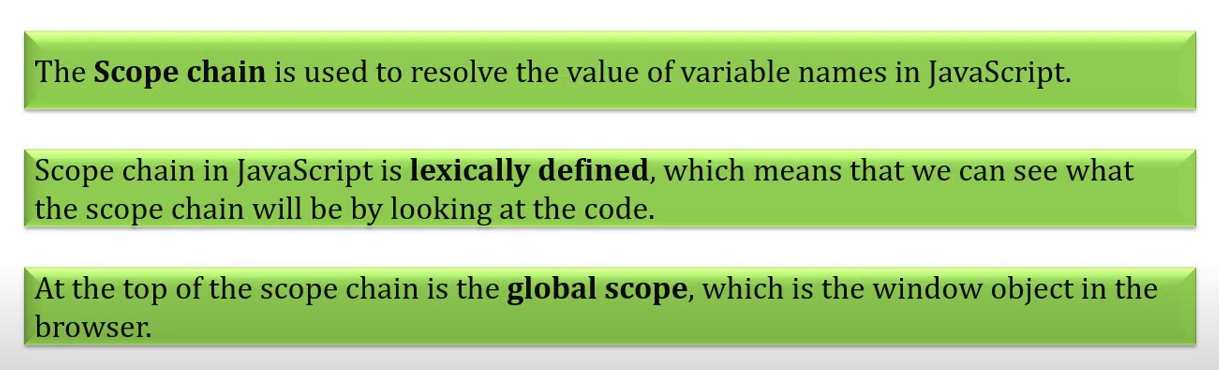
For second console job already define so it console “I am Full Stack”.

Note: The “var = job” which is inside the function so it’s scope is inside the function only not globally.



# Q5. What is Scope Chain & Lexical Scoping?

In JavaScript, the **scope chain** is a mechanism that allows the JavaScript engine **to resolve variable references** during code execution.

It defines the order in which JavaScript looks for variables and functions when executing a piece of code. 

**What is a Scope?**

In JavaScript, a scope refers to the region of the code where a variable is defined and is accessible. There are two types of scopes:

* Global Scope: The global scope is the outermost scope, which contains all the global variables and functions.
* Local Scope: A local scope is created when a function is defined. It contains the variables and functions defined within that function.

Lexical Scoping

A function that is lexically (nested or hierarchy) within another function and get access to the scope of outer function. (Inner function can get access to their outer function variable).



So, the output of the code is :

**Hello guys.. How are you?... Myself Vinod Thapa**

**Explanation**

So , when we call function “First” at that time we get the value of “var b” and same time it calls the function second which is child function of first.

In second function we get the value of “var c”.

At the time of console it print all the variables because function “second is a child fun.. so it can access the variable of it’s parent function and the first function access the variable of global scope”.

**So, in this concept we can see the scope chaining and the lexical scope**

# Q6. JavaScript this Keyword

In **JavaScript, this keyword** refers to the**current context or scope** within which code is executing.

Its value is determined by how a function is called, and it can dynamically change depending on the invocation context.

The **this keyword** refers to different objects depending upon how it is used:

* When used within a method of an object, this points to that object.
* When used by itself, this points to the global object.
* Within a function, this typically points to the global object.
* In a function under strict mode, this becomes undefined.
* During an event, this points to the element that triggered the event.
* Methods such as call(), apply(), and bind() can reassign this to any desired object.

## Note

this is not a variable. It is a keyword. You cannot change the value of this.

## **JavaScript this Keyword Examples:**

### **1. Using this** in a Method

const person = {

    name: 'John',

    age: 30,

    greet() {

      console.log('Hello, my name is ' + this.name + ' and I am ' + this.age + ' years old.');

    }

  };

  person.greet(); // Output: Hello, my name is John and I am 30 years old.

**Exercise on This Keyword**

const obj = {

    name: "Abdul",

    Post: "Software Engineer",

    fun : function(){

        console.log(this.name);   // => Abdul

        console.log(this);       // => It reffers current obj

function child(){ // this is a normal function so it always refers ` window object

            var name = "Rahman"

            console.log("chlid function name" + name);  // => Rahman

            console.log(this);  // => Reffers Window

        }

        child();

    }

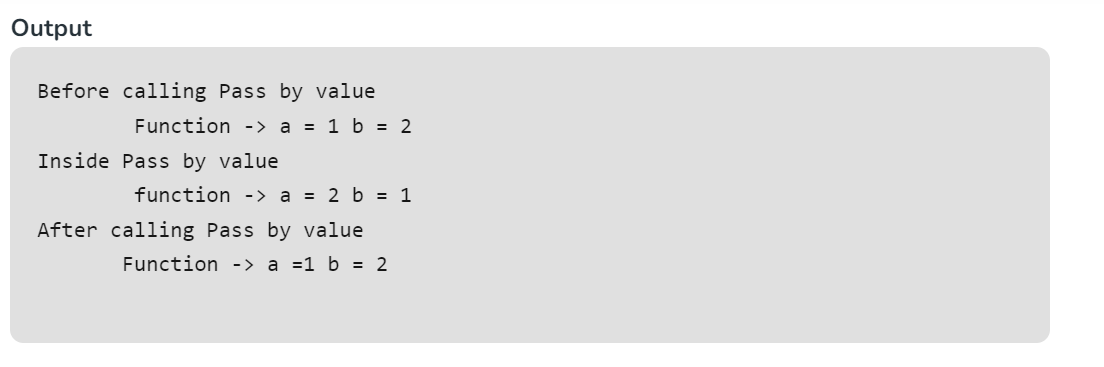
}

obj.fun()

# Q7. Pass By Value & Pass By Reference

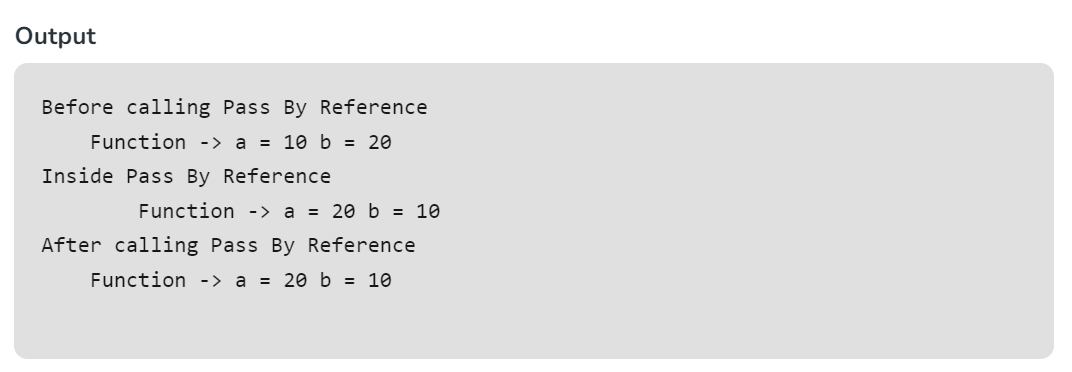
## **Pass By Value**

* In Pass by value, the function is called by directly passing the value of the variable as an argument. So any changes made inside the function do not affect the original value.
* In Pass by value, parameters passed as arguments create their **own copy.** So any changes made inside the function are made to the copied value not to the original value
* function Passbyvalue(a, b) {
* let tmp;
* tmp = b;
* b = a;
* a = tmp;
* console.log(`Inside Pass by value
* function -> a = ${a} b = ${b}`);
* }
* let a = 1;
* let b = 2;
* console.log(`Before calling Pass by value
* Function -> a = ${a} b = ${b}`);
* Passbyvalue(a, b);
* console.log(`After calling Pass by value
* Function -> a =${a} b = ${b}`);

****

## **Pass by Reference**

* In Pass by Reference, Function is called by directly passing the reference/address of the variable as an argument. So changing the value inside the function also change the original value.
* In JavaScript **array and Object**follows pass by reference property.
* In Pass by reference, parameters passed as an arguments does not create its own copy, it refers to the original value so changes made inside function affect the original value.
* function PassbyReference(obj) {
* let tmp = obj.a;
* obj.a = obj.b;
* obj.b = tmp;
* console.log(`Inside Pass By Reference
* Function -> a = ${obj.a} b = ${obj.b}`);
* }
* let obj = {
* a: 10,
* b: 20
* }
* console.log(`Before calling Pass By Reference
* Function -> a = ${obj.a} b = ${obj.b}`);
* PassbyReference(obj)
* console.log(`After calling Pass By Reference
* Function -> a = ${obj.a} b = ${obj.b}`);

****

**Note:** In Pass by Reference, we are mutating the original value. when we pass an object as an arguments and update that object’s reference in the function’s context, that won’t affect the object value. But if we mutate the object internally, It will affect the object .

# Q8. What is First Class Citizen in JavaScript ?

If any programming language has the ability to treat functions as values, to pass them as arguments and to return a function from another function then it is said that  programming language has **First Class Functions** and the functions are called as **First Class Citizens** in that programming language.

 JavaScript has all those abilities or features that are required to be a language having First Class Functions, hence functions are treated as **First Class Citizens.**

**Let’s look at all the abilities of functions being a *First Class Citizen.***

1. **Ability to treat functions as values:**

Functions in JavaScript can be treated as values, i.e. a function can be stored as a value in a variable.

<script>

var greet = function() {

    console.log("Welcome to GeeksforGeeks!");

}

greet();

</script>

Output:

Welcome to GeeksforGeeks!

1. **Ability to pass a function as arguments:**

Functions in JavaScript also has the ability to be passed as arguments to another function.

function teacher(){

    return "Teacher";

}

function student(){

    return "Student";

}

function greet(user){

    console.log("Welcome", user());

}

// Prints "Welcome Teacher"

var message = greet(teacher);

// Prints "Welcome Student"

var message = greet(student);

Output:

Welcome

Teacher

Welcome

Student

1. **Ability to return a function from another function**
2. var greet = function(){
3. return function(){
4. console.log("Welcome to GeeksforGeeks!");
5. }
6. }
7. greet()();

**Output:**

Welcome to GeeksforGeeks!

Note : Here, we use the **double parentheses** to invoke the returned function, hence we use *greet()().*Single parenthesis will call the function itself without invoking its returned function. We can also do it by storing the function in a variable like this-

var func = greet(); func();

**Q9. What is Callback Function and why we use it?**

A callback is a function that is passed as an argument to another function, and is called after the main function has finished its execution.

The main function is called with a callback function as its argument, and when the main function is finished, it calls the callback function to provide a result.

Callbacks allow you to handle the results of an asynchronous operation in a non-blocking manner, which means that the program can continue to run while the operation is being executed.

**Why use Callbacks?**

Callbacks are used to handle asynchronous operations, such as making an API request or reading a file. They allow the program to continue executing other tasks while waiting for the asynchronous operation to complete.

Asynchronous operations are operations that take a significant amount of time to complete, such as network requests, file I/O, and database queries.

If these operations were executed synchronously, the program would freeze and wait for the operation to complete before continuing. This can lead to a poor user experience, as the program would appear unresponsive.

Callbacks allow you to continue executing code while the operation is being executed in the background.

Once the operation has completed, the callback function is called with the result of the operation. This way, you can ensure that the program remains responsive and the user experience is not impacted.

**Real-Life Examples:**

**Loading images on a website:** When you load a website, images can take a while to load, especially if they’re large. If images were loaded synchronously, the website would freeze and wait for each image to load before continuing. With callbacks, you can load the images asynchronously, which means that the website continues to load while the images are being loaded in the background.

**Handling form submissions:** When a user submits a form, it takes time to process the data and send it to the server. If the form submission was executed synchronously, the user would have to wait for the data to be processed and sent before the form can be submitted. With callbacks, you can handle the form submission asynchronously, which means that the user can continue to interact with the form while the data is being processed and sent in the background

// Take callback fun as an parameter and call inside function a

const a = (callback)=>{

    console.log("Function a");

    callback();

}

const b = ()=>{

    console.log("Function b");

}

// call function a and pass function b as a argument that is callback fun

a(b);

Output:

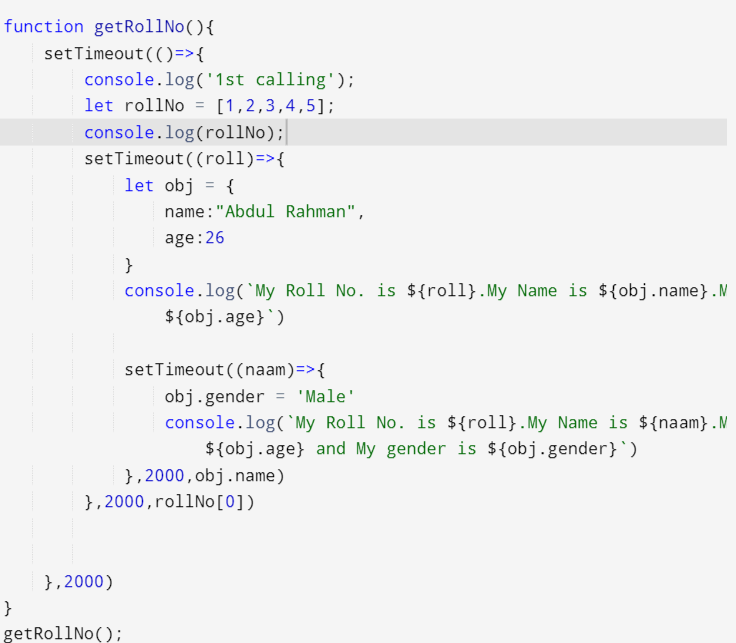
Function a

Function b

**Callback Hell in JavaScript**

Callback hell, also known as "callback hell" or "pyramid of doom," is a phenomenon in JavaScript where nested callbacks create a complex and hard-to-read code structure.

It occurs when a function calls another function, which in turn calls another function, and so on, creating a deep nesting of callbacks.



## **Q.10** **What is a Higher Order Function?**

A **higher-order function** is a function that either:

1. Takes one or more functions as arguments.
2. Returns a function as its result.

function multiplier(factor) {

    return function (x) {

        return x \* factor;

    };

}

const double = multiplier(2);

const triple = multiplier(3);

console.log(double(5));

console.log(triple(5));

**Output**

10

15

**Example**: In this example, `multiplier` is a higher-order function that takes a `factor` as an argument and returns a new function that multiplies any number by that factor. We then use `double` and `triple` to create specialized functions.

## **How to Use Higher Order Functions**

You can use higher order functions in a variety of ways.

* When working with arrays, you can use the map(), reduce(), filter(), and sort() functions to manipulate and transform data in an array.
* When working with objects, you can use the Object.entries() function to create a new array from an object.
* In GUI programming, HOFs are used to handle events such as button clicks, mouse movements, or keyboard inputs. For instance, a button's **onClick** event might take a function as an argument, which is executed when the button is clicked.

A black screen with text

Description automatically generated

## **Approach 1: Function as an Argument:**

This approach involves passing a function (callback) as an argument to another function. The receiving function can then execute the callback, enabling flexible and customizable behavior in JavaScript programs.

A computer screen with text and numbers

Description automatically generated

In this example, the forEach method is a higher-order function that takes a callback function as an argument.

## **Approach 2 : Functions as Return Values**:

Higher-order functions can also return new functions. This is often used for creating specialized functions or closures. For instance, you can create a function factory that generates functions with specific behavior.

function multiplier(factor) {

    return function (x) {

        return x \* factor;

    };

}

const double = multiplier(2);

const triple = multiplier(3);

console.log(double(5));

console.log(triple(5));

**Example**: In this example, `multiplier` is a higher-order function that takes a `factor` as an argument and returns a new function that multiplies any number by that factor. We then use `double` and `triple` to create specialized functions

## **Note : Parameters:**

* **higherOrderFunction:** Takes a callback function, executes it, and performs operations.
* **callback:** A function passed as an argument, executed inside higherOrderFunction.
* **callbackFunction():** Logs “Callback function is executed.”
* **Invocation:** Calls higherOrderFunction(callbackFunction), executing callbackFunction within higherOrderFunction.

**Benefits of Higher-Order Functions**

Higher-order functions provide several benefits, including:

* **Modularity**: They allow you to break down complex operations into smaller, reusable functions.
* **Flexibility**: They enable you to pass different functions as arguments, or return different functions as results.
* **Code reuse**: They promote code reuse by allowing you to write functions that can be used in multiple contexts.

## **Q.11 Differences between First-Order Function and Higher-Order Function**

| **First-Order Function** | **Higher-Order Function** |
| --- | --- |
| Function is treated as a variable that can be assigned to any other variable or passed as an argument. | The function receives another function as an argument or returns First-order a new function or both. |
| The “first-class” concept only has to do with functions in programming languages. | The “higher-order” concept can be applied to functions in general, like functions in the mathematical sense. |
| The presence of the First-class function implies the presence of a higher-order function.  First-order functions operate on data  First-order functions are limited to a specific task | The presence of a Higher-order function does not imply the presence of a First-order function.  Higher-order functions operate on functions themselves.  Higher-order functions can be composed together to perform more complex tasks. |

# Q.12 Immediately Invoked Function Expressions (IIFE)

Immediately Invoked Function Expressions (IIFE) are JavaScript functions that are executed immediately after they are defined.

They are typically used to create a local scope for variables to prevent them from polluting the global scope.

**Syntex**

(function (){   
// Function Logic Here.   
})();

**Example**

(**function**() {

    // IIFE code block

**var** localVar = 'This is a local variable';

    console.log(localVar); // Output: This is a local variable

})();

## **Use Cases Of IIFE**

* Avoid polluting the global namespace.
* To create [closures in JavaScript](https://www.geeksforgeeks.org/closure-in-javascript/).
* IIFE is used to create private and  public variables and methods.
* It is used to execute the[async and await function](https://www.geeksforgeeks.org/async-await-function-in-javascript/).
* It is used to work with [require function](https://www.geeksforgeeks.org/javascript-importing-and-exporting-modules/).

# Q.13 Closure in JavaScript

A closure is a combination of a function and a lexical environment within which that function was declared.

In other words, a closure gives you access to an outer function's scope from an inner function**.**

**OR**

A closure in JavaScript is a function that has access to its own scope and the scope of its parent functions, even when the parent functions have returned.

This allows the closure to "remember" the values of variables from its parent scopes, making it a powerful tool for creating modular and reusable code.

The return statement does not execute the inner function. Functions executed only when followed by (), but rather than return statement returns the entire body of the function.

Lexical Scope Example

function init() {

    var name = "Mozilla"; // name is a local variable created by init

    function displayName() {

      // displayName() is the inner function, that forms the closure

      console.log(name); // use variable declared in the parent function

    }

    displayName();

  }

  init();

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the variables of outer functions, displayName() can access the variable name declared in the parent function, init().

**Example Of clouser**

function makeFunc() {

    const name = "Mozilla";

    function displayName() {

      console.log(name);

    }

    return displayName;

  }

  const myFunc = makeFunc();

  myFunc();

Running this code has exactly the same effect as the previous example of the init() function above. What's different (and interesting) is that the displayName() inner function is returned from the outer function before being executed.

## **Common Use Cases**

Closures are useful in various scenarios:

1. **Maintaining State**: Closures help maintain state between events in event-driven JavaScript.
2. **Private Variables**: You can create private variables by enclosing them within a closure.
3. **Callbacks and Asynchronous Code**: Closures are essential for handling callbacks and asynchronous operations.

**Key differences b/w Closure and Lexical Scoping:**

1. **Scope resolution**: Lexical scoping is about resolving variable names to their corresponding values based on their location in the code. Closures are about a function having access to its own scope and the scope of its parent functions.
2. **Function return**: Lexical scoping does not involve returning a function from another function. Closures involve returning a function from another function, which allows the returned function to access the scope of its parent functions.
3. **Memory retention**: Lexical scoping does not imply that the scope of a variable is retained after the function has returned. Closures, on the other hand, allow the scope of a variable to be retained even after the function has returned.

**Q. 14 What is Call, Apply and Bind Method in JavaScript?**

In JavaScript, call, apply, and bind are three methods that allow you to manipulate the context and arguments of a function.

These methods are part of the Function prototype and can be used to change the behavior of a function.

**Call Method**

Call method is use to call a method which is belonging to another object.

Example:

// Objects declaration

let emp1 = {

    name: "A",

    id: "123",

    fun: function(){

        console.log(`My name is ${this.name} and my id is ${this.id}`)

    }

}

let emp2 = {

    name: "B",

    id: "456",

}

 emp1.fun.call(emp2); // here we call the em1 object function by using call method with the help of emp 2 object

**How to take parameter value using call method**

// Objects declaration

let emp1 = {

    name: "A",

    id: "123",

    fun: function(post){

        console.log(`My name is ${this.name} and my id is ${this.id} and my post is ${post}.`)

    }

}

let emp2 = {

    name: "B",

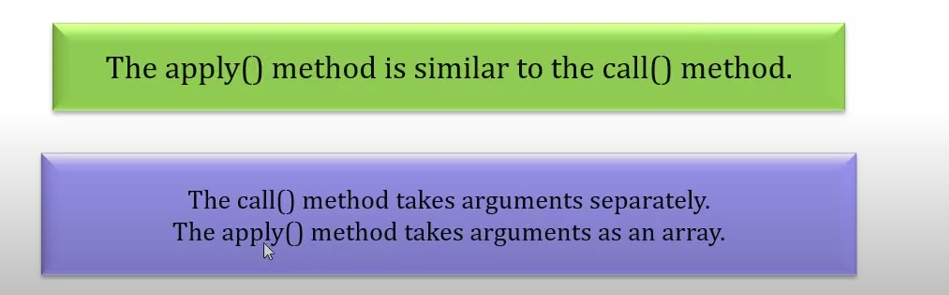
    id: "456",

}

// Pass an argument in call method

 emp1.fun.call(emp2, "Senior Analyst");

**Apply Method**

****

**Example**

let student = {

    details: function (section, rollnum) {

        return this.name + this.class

            + " " + section + rollnum;

    }

}

let stud1 = {

    name: "Dinesh",

    class: "11th",

}

let stud2 = {

    name: "Vaibhav",

    class: "11th",

}

// PAssing an argument in form of array

let x = student.details.apply(stud2, ["A", "24"]);

console.log(x);

**Output:**

Vaibhav

11th

A

24

**Bind Method**

In [JavaScript](https://www.geeksforgeeks.org/introduction-to-javascript/) function binding happens using the [**Bind()** method](https://www.geeksforgeeks.org/explain-call-apply-and-bind-methods-in-javascript/).

With this method, we can bind an object to a common function, so that the function gives different results when needed.

otherwise, it gives the same result or gives an error while the code is executing.

We use the [**Bind()** method](https://www.geeksforgeeks.org/explain-call-apply-and-bind-methods-in-javascript/) to call a function with ‘**this’** value.

**Example 1:** In this example, this keyword binds the name variable to the function. It is known as default binding. this keyword refers to **geeks** object.

let geeks = {

    name: "ABC",

    printFunc: function () {

        console.log(this.name);

    }

}

geeks.printFunc();

**Output**

ABC

**Example 2:**In this example, the binding of this is lost, so no output is produced.

let geeks = {

    name: "ABC",

    printFunc: function () {

        console.log(this.name);

    }

}

let printFunc2 = geeks.printFunc;

printFunc2();

**Output**

undefined

**Example 3:** In this example, we are using the **bind()** method in the previous example. The **bind()** method creates a new function where **this** keyword refers to the parameter in the parenthesis. This way the **bind()** method enables calling a function with a specified **this** value.

let geeks = {

    name: "ABC",

    printFunc: function () {

        console.log(this.name);

    }

}

let printFunc2 = geeks.printFunc.bind(geeks);

//using bind()

// bind() takes the object "geeks" as parameter//

printFunc2();

**Output**

ABC

**Example 4:** In this example, there are 3 objects, and each time we call each object by using the **bind()**method.

//object geeks1

let geeks1 = {

    name: "ABC",

    article: "C++"

}

//object geeks2

let geeks2 = {

    name: "CDE",

    article: "JAVA"

}

//object geeks3

let geeks3 = {

    name: "IJK",

    article: "C#"

}

function printVal() {

    console.log(this.name + " contributes about " +

                this.article + "<br>");

}

let printFunc2 = printVal.bind(geeks1);

//using bind()

// bind() takes the object "geeks1" as parameter//

printFunc2();

let printFunc3 = printVal.bind(geeks2);

printFunc3();

let printFunc4 = printVal.bind(geeks3);

printFunc4();

//uniquely defines each objects

**Output**

ABC contributes about C++<br>

CDE contributes about JAVA<br>

IJK contributes about C#<br>

# Q.15 JavaScript Promise

# 

**Syntax:**

// Create a constructor

let promise = new Promise(function(resolve, reject){  
 //do something  
});

**Parameters**

* The promise constructor takes only one argument which is a callback function
* The callback function takes two arguments, *resolve* and *reject*
  + Perform operations inside the callback function and if everything went well then call resolve.
  + If desired operations do not go well then call reject.

**A Promise has three states:**

* ***Pending****: Initial state, where the operation has not started yet.*
* ***Fulfilled****: The operation was successful, and the promise has a value.*
* ***Rejected****: The operation failed, and the promise has a reason for the failure.*

**Example 1:**In this example we create a promise comparing two strings. If they match, resolve; otherwise, reject. Then, log success or error accordingly. Simplifies asynchronous handling in JavaScript.

* let promise = new Promise(function (resolve, reject) {
* const x = "geeksforgeeks";
* const y = "geeksforgeeks"
* if (x === y) {
* resolve();
* } else {
* reject();
* }
* });
* promise.
* then(function () {
* console.log('Success, You are a GEEK');
* }).
* catch(function () {
* console.log('Some error has occurred');
* });

**Output**

Success, You are a GEEK

**Promise Consumers**

Promises can be consumed by registering functions using ***.then*** and ***.catch***methods.

### **Promise Method:**

Promises have several methods that can be used to handle the outcome of the operation:

* **then()**: Attaches a callback to handle the fulfilled value.
* **catch()**: Attaches a callback to handle the rejected reason.
* **finally()**: Attaches a callback to handle the outcome of the operation, regardless of whether it was fulfilled or rejected.

### **1. Promise then() Method:**

**Parameters:** It takes two functions as parameters.

* The first function is executed if the promise is resolved and a result is received.
* The second function is executed if the promise is rejected and an error is received. (It is optional and there is a better way to handle error using .catch() method

**Syntax:**

.then(function(result){  
 //handle success  
}, function(error){  
 //handle error  
})

**Example 2:**This example shows how the then method handles when a promise is resolved

let promise = new Promise(function (resolve, reject) {

  resolve('Geeks For Geeks');

})

promise

  .then(function (successMessage) {

      //success handler function is invoked

      console.log(successMessage);

  }, function (errorMessage) {

      console.log(errorMessage);

  });

**Output**

Geeks For Geeks

**Example 3:**This example shows the condition when a rejected promise is handled by second function of then method

let promise = new Promise(function (resolve, reject) {

  reject('Promise Rejected')

})

promise

  .then(function (successMessage) {

      console.log(successMessage);

  }, function (errorMessage) {

      //error handler function is invoked

      console.log(errorMessage);

  });

**Output**

Promise Rejected

### **2. Promise catch() Method:**

[**Promise catch() Method**](https://www.geeksforgeeks.org/javascript-promise-catch-method/)is invoked when a promise is either rejected or some error has occurred in execution.

It is used as an Error Handler whenever at any step there is a chance of getting an error.

**Parameters:** It takes one function as a parameter.

* Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler) )

**Syntax:**

.catch(function(error){  
 //handle error  
 })

**Examples 4:**This example shows the catch method handling the reject function of promise.

let promise = new Promise(function (resolve, reject) {

  reject('Promise Rejected')

})

promise

  .then(function (successMessage) {

      console.log(successMessage);

  })

  .catch(function (errorMessage) {

      //error handler function is invoked

      console.log(errorMessage);

  });

**Output**

Promise Rejected

**Finally in Promiss**

****

**Benefits:**

Promises provide several benefits, including:

* **Improved code readability:** Promises make it easier to write and read asynchronous code.
* **Error handling**: Promises provide a way to handle errors in a centralized manner.
* **Chaining operations**: Promises allow you to chain multiple operations together, making it easier to handle complex asynchronous operations.

**Common Pitfalls:**

* Callback hell: Promises can lead to callback hell if not used properly.
* Unhandled rejections: Promises can lead to unhandled rejections if not handled properly.

**How do Promise.all() and Promise.race() work?**

* **Explanation Expected:**
  + **Promise.all()**: Resolves when all promises in an array are resolved, or rejects when any one promise is rejected.
  + **Promise.race()**: Resolves when the first promise in the array resolves or rejects when the first promise rejects.

# Q.16 Async and Await in JavaScript

**Async/await** is a feature in JavaScript that allows you to **work with asynchronous code** in a more synchronous-like manner, making it easier to write and understand asynchronous code.

They were introduced in ECMAScript 2017 and have since become a popular way to handle asynchronous operations.

There are two terms

1. Async : **Async always use with Functions** which return a promise.

2. Await: **Await Keyword** is used only in Async Functions to wait for promise.

**Async Syntax**

async function myFunction() {

  return "Hello";

}

### **Async Function Example**

Here, we will see the basic use of async in JavaScript.

const getData = async () => {

  let data = "Hello World";

  return data;

}

getData().then(data => console.log(data));

**Output**

Hello World

### **Explanation**

Here, an asynchronous function getData is declared using the arrow function syntax with the async keyword. Inside this function, a variable data is initialized with the string “Hello World”. Then, the function returns this data.

## **Await Keyword**

**Await**is used to wait for the promise. It could be used within the async block only.

The **await** keyword is used to pause the execution of an async function until a promise is resolved or rejected. When you use **await** with a promise, it will pause the execution of the function until the promise is settled.

### **Await Syntax**

let value = await promise;

### **Await Example**

This example shows the basic use of the await keyword in JavaScript.

const getData = async () => {

  let y = await "Hello World";

  console.log(y);

}

console.log(1);

getData();

console.log(2);

**Output**

1

2

Hello World

### **Explanation**

Here, an asynchronous function getData is defined using the arrow function syntax with the async keyword. Inside this function, there’s an await expression, which pauses the execution of the function until the promise is resolved. However, in this case, the awaited value is a string literal “Hello World”, which is not a promise, so it will be immediately resolved.

**Note**

* *The****async****keyword transforms a regular JavaScript function into an asynchronous function, causing it to return a Promise.*
* *The****await****keyword is used inside an async function to pause its execution and wait for a Promise to resolve before continuing.*

## **Async/Await Example**

function resolveAfter2Seconds() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve('resolved');

    }, 2000);

  });

}

async function asyncCall() {

  console.log('calling');

  const result = await resolveAfter2Seconds();

  console.log(result);

  // Expected output: "resolved"

}

asyncCall();

**Output**

"calling"

"resolved" // after 2 sec it will display

**Q.17 What is Prototype in javascript**

**Prototype property** is basically an object (also known as Prototype object).

JavaScript is a prototype based language, so, whenever we create a function using JavaScript, JavaScript engine adds a “prototype property” inside a function), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties.

**OR**

It is a mechanism by which javascript inherits features from one another.

A prototype is an object that is associated with a function or an object. It is used to define the properties and behavior of an object. When you create an object, it inherits the properties and behavior from its prototype.

**Consider the example below:-**

let users = {

            getFulName : function(){

                return this.name + " " + this.lastName;

            },

            getAge: function(){

                let age = new Date().getFullYear()  - this.birth;

                return age;

            }

        }

        let student = {

            name:"Abdul Rahman",

            lastName: "Khan",

            birth: 1998

        }

        let teacher = {

            name:"XYZ",

            lastName: "Demo",

            birth: 1980

        }

        student.\_\_proto\_\_ = users;

        teacher.\_\_proto\_\_ = users;

        console.log(student.getAge());

        console.log(teacher.getAge());

**What happen we we load property direct in our object**

 let student = {

            name:"Abdul Rahman",

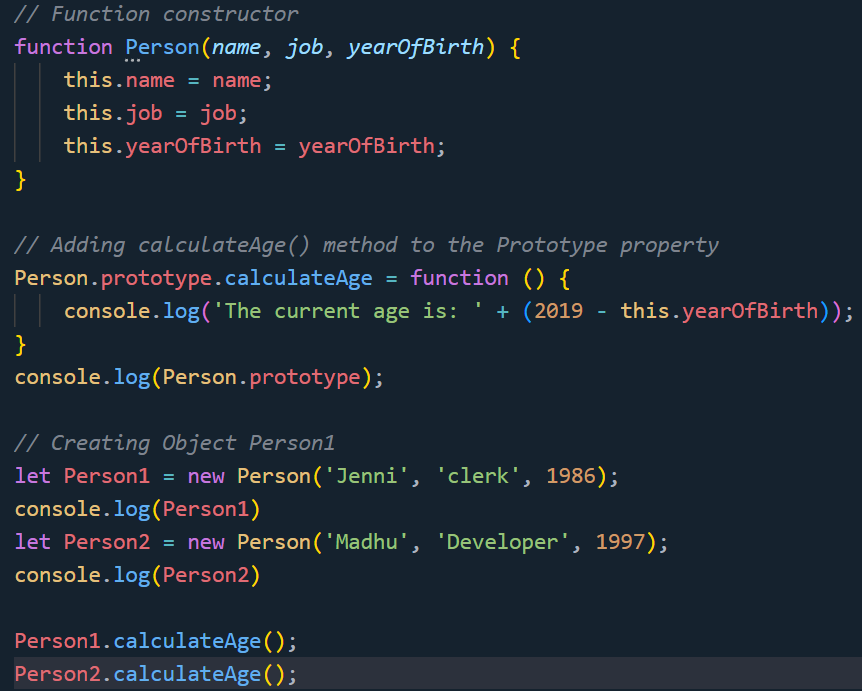
            lastName: "Khan",

            birth: 1998,

            getAge: users.getAge    // If we load direct property in object so our application will slow

                                    // but if we load the property using \_\_proto\_\_ on that time the property will load when we call it.

        }

****

**Prototype Chain**

The prototype chain is a series of prototypes that are linked together through the \_\_proto\_\_ property.

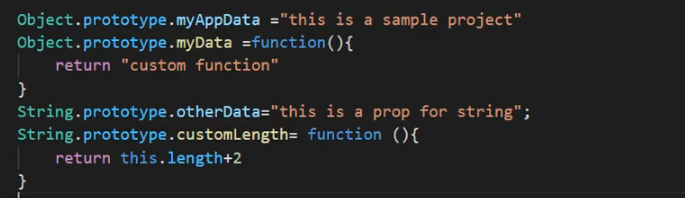
When you access a property or method on an object, JavaScript traverses the prototype chain to find the property or method.

**A computer screen shot of text

Description automatically generated**

In this example, **Dog** is a function that creates an object that inherits from **Animal**. The **Dog.prototype** object is created using **Object.create(Animal.prototype)**, which sets up the prototype chain. When you access the **sound** method on **myDog**, JavaScript traverses the prototype chain to find the method on **Animal.prototype**.

**How to create own prototype**

****

**Q18 . What is event delegation in JavaScript?**

* Explanation Expected: Event delegation is the practice of attaching a single event listener to a parent element rather than multiple listeners to each child element. This allows handling events for dynamically added elements.

**ES6 Features**