What is JavaScript

JavaScript (js) is a light-weight object-oriented programming language which is used by several websites for scripting the webpages. It is an interpreted, full-fledged programming language that enables dynamic interactivity on websites when applied to an HTML document.

## History of JavaScript

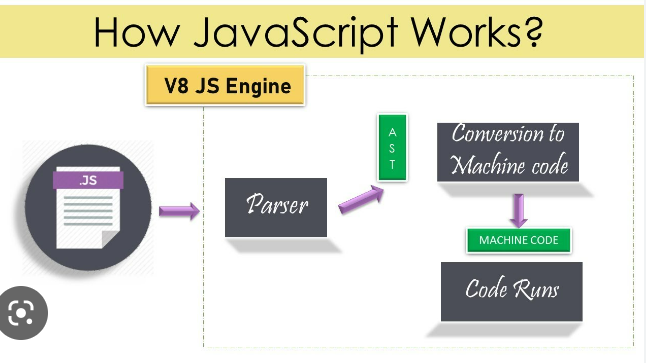
In 1993, **Mosaic**, the first popular web browser, came into existence. In the **year 1994**, **Netscape** was founded by **Marc Andreessen**. He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be provided to HTML to make web designing easy for designers and part-time programmers. Consequently, in 1995, the company recruited **Brendan Eich** intending to implement and embed Scheme programming language to the browser. But, before Brendan could start, the company merged with **Sun Microsystems** for adding Java into its Navigator so that it could compete with Microsoft over the web technologies and platforms. Now, two languages were there: Java and the scripting language. Further, Netscape decided to give a similar name to the scripting language as Java's. It led to 'Javascript'. Finally, in May 1995, Marc Andreessen coined the first code of Javascript named '**Mocha**'. Later, the marketing team replaced the name with '**LiveScript**'. But, due to trademark reasons and certain other reasons, in December 1995, the language was finally renamed to 'JavaScript'. From then, JavaScript came into existence.

Features of JavaScript

There are following features of JavaScript:

1. All popular web browsers support JavaScript as they provide built-in execution environments.
2. JavaScript follows the syntax and structure of the C programming language. Thus, it is a structured programming language.
3. JavaScript is a weakly typed language, where certain types are implicitly cast (depending on the operation).
4. JavaScript is an object-oriented programming language that uses prototypes rather than using classes for inheritance.
5. It is a light-weighted and interpreted language.
6. It is a case-sensitive language.
7. JavaScript is supportable in several operating systems including, Windows, macOS, etc.
8. It provides good control to the users over the web browsers.

**1.Js Engine Working principle**

****

-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. Nowadays, most of the modern web browsers

support JavaScript and have their JavaScript engines.

For example, Google Chrome has its own JavaScript

engine called V8.

-Now let's see how the JavaScript engine handles and runs .js code.

-In this case, we have used a chrome browser

to run our program that has the "V8" JavaScript engine,

which is also used for creating the Node.js

-As we already know, JavaScript is an interpreted language that means it gets executed in line by line manner

(or which means the JavaScript engine converts the Js code line by line and runs in the same manner instead of converting the whole program once).

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.

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).

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.

2.How to include JavaScript in an HTML document?

-we have to ways

a.internal

-1.inside html document create script tag and write js program

b.external

1. create html file

2.Add script tag in html document.

3.Script tag is for attaching js file.

4.use js for browser.

3.what is static page?

-A static Web page is a page that is built using HTML code and features the same presentation and content,

regardless of user identity or other factors.

4.What is dynamic page?

-Using JavaScript we can create the dynamic webpage which can perform any action according to instruction and then display it dynamically on the webpage.

5.What is single page application(modern approach)?

-A Single Page Application (SPA) is a single web page, website,

or web application that works within a web browser and loads just a single document.

It does not need page reloading during its usage, and most of its content remains the same while

only some of it needs updating. When the content needs to be updated, the SPA does it through JavaScript APIs.

6.What is multi-page application(traditional approach)?

-As the name suggests, these are web applications that have more than one page.

-Multipage applications load their pages every time a user clicks on different links.

-MPAs are separately built multiple pages that combine together and make a website.

1.Tokens

Smallest unit of any programming language.

Keywords : a predefined reserved word which js engine will understand.

a. Every keyword must be lowercase.

b. A keyword cannot be used as identifier.

c. Eg : if , for , let , const ,null, var

2.Identifiers : name provided by programmer to the components of js like variable, function, class etc.

Rules for identifiers

i. An identifier cannot start with number.

ii. Except $ and \_ no other special character is allowed

iii. We cannot use keywords as identifiers.

3. Literals/values: The data which is used in js code/program.

Operator : perform an action on data

Some popular engines

1. V8 for google

2. SpiderMonkey for Mozilla

3. JavaScriptCore Safari

4. Chakra for Internet Explorer

**Characteristics of Javascript**

It is purely object oriented language (based on objects for storing most kind of data)

Interpreted language., not compiled language

• Line by line execution, like first it will check if it correct it execute and check next line

javascript uses just in time (JIT)

Javascript is synchronous in nature : single threaded architecture have one stack for execution.

**Implementation of JS**

1.internal within html document by suing script tag

2.External we need to create one new file extension should be .js and link with script src .

a) We can create a separate file for js with extension.js

b)Link the js file with html page using src attribute of script tag.

<html>

<head>

<title>js separate</title>

</head>

<body>

<h1>linking js</h1>

<!-- <script src="path/filename.js"></script> -->

<script src="../js/js1.js"></script>

</body>

</html>

**Typeof**

It is a keyword use as unary operator to identify the type of data.

typeof - will check the value belongs to which category

(typeof 10);

Syntax

typeof value

Program

console.log(typeof "samsung");//string

console.log(typeof `samsung`);//string

console.log(typeof 'samsung');//string

console.log(typeof 1);//number

console.log(typeof null);//object

console.log(typeof undefined);//undefined

console.log(typeof true);//boolean

console.log(typeof false);//boolean

Note

Typeof null is never null, it is considered as objec**t**

**Types of values or datatypes in javascript:**

1.Number

a. The numbers between -( 2^53 -1) and ( 2^53 -1)---number. Eg : 1----number type

2. String

3.Boolean

a. True 1

b. False 0

4. Null : it is keyword , it define value

5.Undefined : it is keyword , it define value

a. When we declare a variable in js, js engine implicitly assign undefined value to it.

6. Object

Can we run js outside the browser?

We can run js outside browser with help of node.

**Node**

Node is a bundle(extra layer) of google v8 engine and built in methods using c++.

It serves as environment to run js without help of browser/outside browser.

We use js(node) for business logic.

This invention helped js to gain its popularity in usage as a backend language.

With the help of node we can run without browser.

**variables**

A named block of memory which is used to store the value is known as variable.

(container to store data)

Js is not strictly type language as we don’t have to specify what type of data we want to store. It is dynamic type

language it will understand while execution .It is not necessary to specify type of data during variable declaration.

In a variable we can store any type of value.

It is mandatory to declare a variable before using.

We create variable using variable declaration followed by identifier and we can store anything

**Syntax for variable declaration**

var a; // declaration stmt

a=10; // initialization / assignment

console.log(a);

let b;

b=20;

console.log(b);

const c=30;

console.log(c);

**Understanding the scope of variable**

**Scope**: The visibility of a member to access it is known as scope.

**Global scope:** it has highest accessibility can be access anywhere.

**Block scope :** The visibility of a member is only within the block where it is declared. A member with

block scope can be used only inside the block where it is declared, it cannot be used outside.

**Note:** The variable declared with let and const have block scope.

Eg1:

{

let a=10;

console.log(a);//can be used

}

console.log(a);//cannot be used

Eg2:

{

const a=10;

console.log(a);//can be used

}

console.log(a);//cannot be used, error variable not defined

Eg3:

{

var a=10;

console.log(a);//can be used

}

console.log(a);//can be used

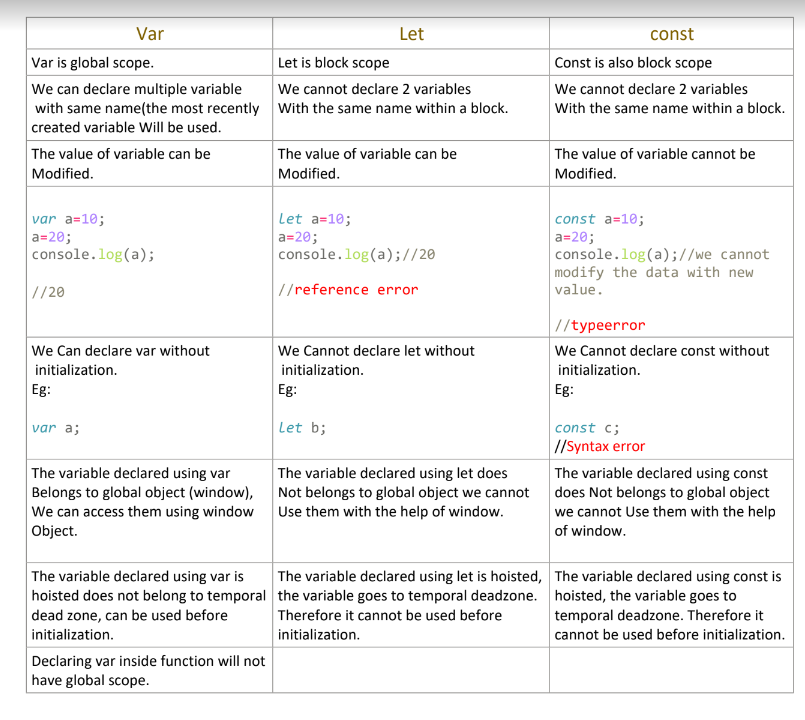
**Var**

• If we declare a variable using var it has global scope, even inside block it acts has same

**let**

• A variable creating using let we can use only inside block where it is declared.

• We cannot declare more than 1 with same name in same scope.



**Browser**

It’s a application which is running over machine or processor .

1. He can understand html, css behaves like compiler and interpreter which convert all these into machine language.

2. Browser give environment to run js instruction.

3. In browser we have many sub-application.

In browser we have js engine(sub application)

a. It is used to translate js instruction to machine level language.

4.

5. Every browser will have a js engine to run js code. Therefore the browser become an environment to run js

6. Browser has a default console.

**Hoisting**

Js allows a programmer to use a member (variable) before the declaration statement. This characteristic is known as hoisting.

What makes js hoisting?

It runs js in 2 phases

1. Allocates memory variables and assign value(undefined)

2. Instruction get executed (top to bottom).

Since the declaration is already done we can use.

Understanding execution in js

1. Every time when js engine runs a js code , it will first create a Global Execution context.

i. Global execution context is a block of memory

Functional area /

Execution area

The global execution context has 2 parts

ii. Variable area.

iii. Functional area or execution area.

**Typecasting**

Implicit typecasting/type conversion : js engine convert one type of data to another type implicitly of

data when wrong type of value is used in expression .

Number zero (0), null , NaN, empty string(''), undefined all these values are considered as false

when they are converted to boolean.

1.In js everything rather above consider as true.

console.log(100?10:20); //every non zero will consider than 0

console.log('hi'?10:20);//

var a=5-'1'; // string converted to number and add

console.log(a);

var b=5+'1'; // string converted to number and concat

console.log(b);

var x=5-'a'; // hence it is converted to a special number NaN

console.log(x);

var y=(typeof (5-'a'));

console.log(y);

2.Any arithmetic operation with NaN result is NaN

Two NaN is never consider as equal

console.log( NaN === NaN); //false

console.log( 10 === NaN); //false

**Logical OR**

Logical OR behaves differently if LHS value or RHS value is non boolean

Step 1: it converts the LHS value to boolean.

Step 2 : if the converted LHS value is true than it returns the original value present in the LHS

Eg

console.log(20 || 10); // so output will be 20

if the converted LHS value is false than it returns the original value present in the RHS

console.log(0 || 10); // so output will be 10

**Explicit Type Casting :**

The process of converting from one type of value to another type of value is

known as explicit type casting.

Case 1 : Conversion of any type to number

Syntax

Number(data-to-be-converted)

i.If a string is valid number we get real number

console.log(Number('123')); // 123

ii.If the string consist any other character then we get NaN as output.

a. console.log( Number ('a')); //NaN

Boolean to Number

console.log(Number (false)); //0

// when we convert false to 0 the result will 0

console.log(Number (true)); //1

// when we convert true to 0 the result will 0

console.log(Number (20>10)); // 1

**Decision Statements**

Decision statement helps to skip a block of instruction when we don’t have favoring situation. Eg: The

instruction of loading home page should be skipped if the entered password is incorrect.

Decision statement of js:

1.If

if(condition)-------> any expression whose result is Boolean

{

}

2.If else

if(condition)-------> any expression whose result is Boolean

{

}

else{

}

3.Else if

var firstnumber=Number( prompt("enter first number"));

var secondnumber=Number( prompt("enter second number"));

var thirdtnumber=Number( prompt("enter third number"));

var fourthnumber=Number( prompt("enter fourth number"));

if(firstnumber<secondnumber && firstnumber<thirdtnumber &&

firstnumber<fourthnumber )

{

console.log(` ${firstnumber} is smallest` );

}

else if( secondnumber<thirdtnumber && secondnumber<fourthnumber )

{

console.log(` ${secondnumber} is smallest`);

}

else if(thirdtnumber<fourthnumber)

{

console.log(` ${thirdtnumber} is smallest`);

}

else {

console.log(` ${fourthnumber} is smallest`);

**Switch**

Switch is keyword and we pass value not boolean condition

Switch is faster than if else

Case followed by value or expression followed by : and then statements

Default is optional

And we can write default anywhere , not necessary to write at end.

Syntax

switch(value)

{

case value : {

statement;

}

case value : {

statement;

}

.

.

.

default : {

statement;

}

}

1.A case blocks gets executed if the value passed to switch matches with value present in case

2.When a case is favorable the case blocks gets executed as well as all the blocks present below in

the switch gets executed.

3. We can have only 1 default inside switch.

4.Default can be written anywhere in switch.

switch(1)

{

case 1 : {console.log(`case 1`);}

default : {console.log(`case 1`);}

case 2 : {console.log(`case 1`);}

case 3 : {console.log(`case 1`);}

}

**Break**

• It is a control transfer statement.

• It can be used inside either in switch block or loop block only.

* When a break statement is encountered the control list transferred outside the current switch or

loop block.

switch(6)

{

case 1 : {console.log(`case 1`);}

break;

Literal : the value will not change

True and false are literal

default : {console.log(`case 1`);}

case 2 : {console.log(`case 1`);}

break;

case 3 : {console.log(`case 1`);}

**Looping**

It is also called iteration.

The process of executing a instruction /block of instruction repeatedly multiple times.

Note: when we design a loop it is a responsibility of a programmer to break the loop after achieving the desire task, if not the

program get into infinite loop statement.

If we don’t break loop it will be infinite and control will not come out of loop.

Condition is use to break loop after some certain

Loop statements

1.While

a. Syntax

while(condition){

statement to be repeated;

}

2.Do-while :

The do-while loop loops through a block of code once, then the condition is evaluated. If the condition is

true the statement is repeated as long as specified condition is true.

a. number of iteration in do while is 1

Syntax

do{

// statement;

}

while(condition);

**Note:**

a. in do while loop the body of the loop is executed first then the condition is evaluated.

b. If the condition evaluated is true the body of the loop inside the do statement is executed again.

c. The statement inside the do statement is executed again.

d. This process continue until the condition evaluates to false. Then the loop stops.

3. For

4. For-in etc

**Functions**

• Function is a block of instruction which is used to perform a specific task

• A function get executed only when it called.

• The main advantage of function is we can achieve code reusability.

• To call a function we need its reference and ()

**Note**: in javascript functions are beautiful, every function is nothing but an object.

Syntax to create a function.

Generally we can create a function in 5 ways:

1. Function declaration statement(function statement).

2. Function expression.

Function declaration / statement(function statement).

Syntax :

function identifier ([list\_of\_parameter,...])

{

statements;

}

1.

Note:

i. Function is object

ii. Name of function is variable which holds the reference of function object.

iii. Creating a function using function statement supports function hoisting.

iv. Therefore we can also call a function before function declaration.

v. when we try to log function name the entire function defination is printed.

console.log('start');

console.log(test);

function test(){

console.log('Hello');

}

console.log('start');

test - it is a function variable - it will return body of function

test () - invoking a function.

To call a function - function\_name();// we can also pass data

Function\_name(arguments\_list,.....);

console.log('start');

function test(){

console.log('Hello');

}

test();

test();

test();

console.log('start');

parameter(placeholder: variable to hold data when function is called)

console.log('start');

function test(a){

console.log('Hello');

console.log(a);

}

test(10); argument : values which are passed.

console.log('start');

Parameters

The variables declared in the function defination is known as parameters.

The parameters have local scope (can be used only inside function body).

Parameters are used to hold the values passed by calling a function.

Eg:

function sum(a, b){

console.log(a + b)

}

// a and b are variables local to the function sum

Arguments

The values passed in the method call statement is known as arguments.

Note : An argument can be a literal, variable or an expression which gives a results.

Eg 1:

sum(10,20); // 10,20 are literals used as arguments

Eg 2:

sum(-10+3,-20);//-27

Eg 3:

sum(a ,b); //a and b are variable used as argument

**return**

1. It is a keyword used as control transfer statement in a function.

Return will stop the execution of the function and transfer control along with data

to the caller.

2.

function toMeters(cms){

return (cms/100);

}

var cms =2546;

console.log(toMeters(cms));

var m =toMeters(cms );

console.log(m);

The ability of a function to use as value is called first class function

**2. Function as Expression**

Syntax :

var/let/const identifier = function (){

}

The function is used as value

Disadvantage :

• function is not hoisted

• We cannot use a function before declaration.

a();// error

var a = function (){ /\* a has address of function\*/

console.log("fun");// will get entire function definition

}

In the above example function is not hoisted instead variable is

hoisted and assigned with default value undefined. therefore type of

a is not function it is undefined.

**This**

It is a keyword used as variable.

It holds the address of global window object.

Therefore with the help of this variable we can use members of global window object.

Whenever we call a function a new execution context is created. Inside that all local

variable declare inside will be there and 1 more "this" will be there which is different from

this

Inside this it have address of which function belongs to.

a.In javascript 'this' is a property of every function.(every function will have this

keyword)

b. Generally this contains the current execution context to which the function belongs.

**3.Arrow Functions**

• Arrow function was introduce from ES6 OF JS.

• The main function of using arrow function is to reduce syntax

Syntax

( parameter\_list,...) => {}

Note:

1. Parameter is optional.

2. If function has only one statement , then block is optional.

3.it is mandatory to create a block if return keyword is used.

Error if we write like this

const c =(n1+n2) => return n1+n2;

console.log(c(10,20));

Correct solution

const c =(n1, n2) => { return n1+n2};

console.log(c(10,20));

Operation taking parameter using arrow function

**4.IIFE-(Immediate invocation function)**

When a function is called immediately as soon as the function object is created.

Steps to achieve

• Treat a function like a expression by declaring in pair of bracket.

Add another pair pf braces next to it which behaves like function call statement.

○ Eg1

(function abc(){

console.log("Hi");

})();

○ Eg2

let a= ( ()=>{console.log("hello"); return 10})();

console.log(a);

**5.Anonymous function**

The function declare without any name which is called as anonymous function

Syntax:

function (){

instruction

}

**Js Engine Working principle?**

-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. Nowadays, most of the modern web browsers

support JavaScript and have their JavaScript engines.

For example, Google Chrome has its own JavaScript

engine called V8.

-Now let's see how the JavaScript engine handles and runs .js code.

-In this case, we have used a chrome browser

to run our program that has the "V8" JavaScript engine,

which is also used for creating the Node.js

-As we already know, JavaScript is an interpreted language that means it gets executed in line by line manner

(or which means the JavaScript engine converts the Js code line by line and runs in the same manner instead of converting the whole program once).

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.

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).

Step 3: Conversion to Machine code

-Once the Abstract Syntax Tree is created by the parser, the JavaScript engine converts the

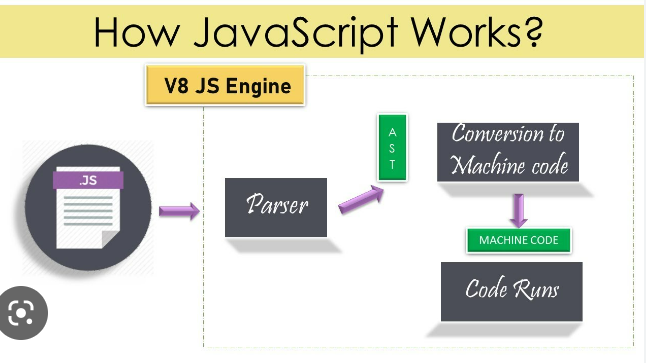
JavaScript code into the machine code (or in the language that machine can understand).

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.



FOR IN LOOP

The for...in statements combo iterates (loops) over the properties of an object.

The code block inside the loop is executed once for each property.

SYNTAX

for (const key in object) {

//instructions

}

Examples

Iterate (loop) over the properties of an object:

const person = {fname:"John", lname:"Doe", age:25};  
let text = "";  
for (let x in person) {  
  text += person[x] + " ";  
}

FOR OF LOOP

The for...of statements combo iterates (loops) over the values of any iterable.

The code block inside the loop is executed once for each value.

SYNTAX:

for (const iterator of object) {

}

Examples

Iterate (loop) over the values of an array:

let text = "";  
const cars = ['BMW', 'Volvo', 'Mini'];  
for (let x of cars) {  
  text += x + " ";  
}

## Arrays

Array is a block of memory which is used to store multiple values of hetrogeneous type .

Creation of arrays

In js array is an object.

We can create array object in three different ways.

1. Using array literal.
   1. Syntax

*let* arr = [value1, value2, ];

Eg:

*let* arr = [10,20,30]; console.log(typeof arr);

1. By creating an instance of array using new operator.

*let* arr2 = new *Array*();

1. By creating an instance of array and initializing the elements using array can stopped.

*let* arr = new *Array*(10,20,30);

To access the array element

We can access the array element with the help of array object reference and array operator and index.

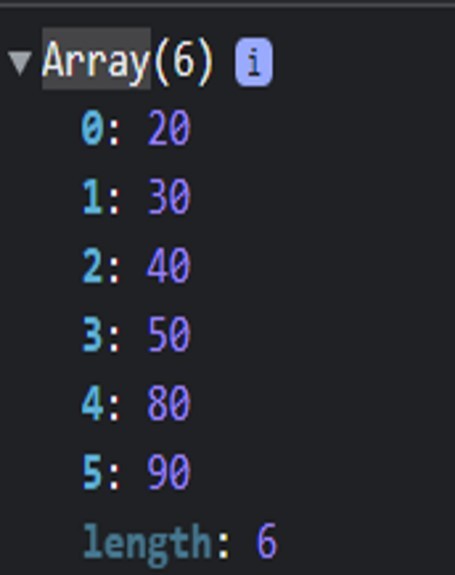
array\_object\_ref[index];

Index

It is a number which starts with 0 and ends with length of the array object -1. Eg:

*let* arr1= [10,20,30];

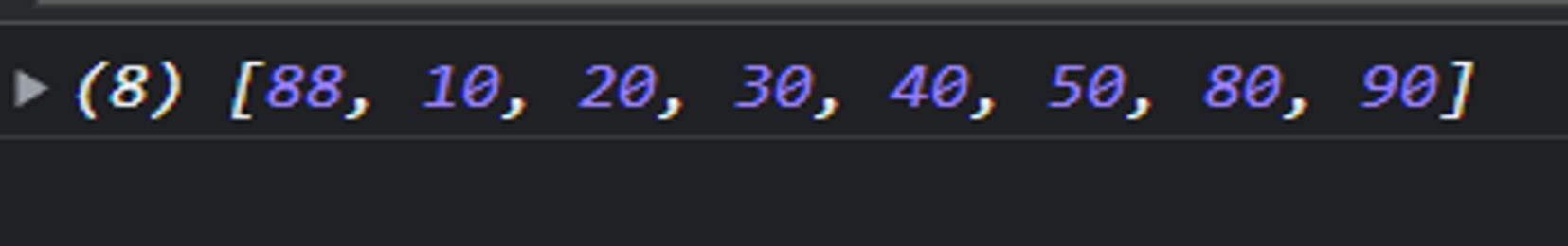
//To print 2nd element, index is 2-1=1 console.log(arr1[1]);

Shift method will help to remove from first index.

*var* arr=[10,20,30,40,50,80,90];

arr.shift(); console.log(arr);

Unshift will add item in first index.



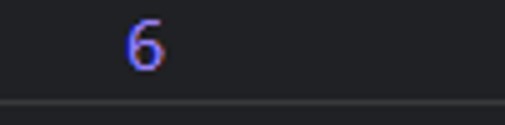
To retrieve element from array we use index.

indexOf()

The indexOf method returns the index of a particular element which is passed to it.

*var* hobbies

=['reading','singing','cycling','sleeping','coding','travellin g','cricket'];

console.log(hobbies.indexOf('cricket')); 

console.log(hobbies.indexOf('eating'));

includes()

The includes method of array weather the search element is present in an array, along with its index value.

Syntax

includes(search value , from index) From index is not compulsory.

Example 1

*var* hobbies

=['reading','singing','cycling','sleeping','coding','travellin

g','cricket']; console.log(hobbies.includes('cricket'));

Output : true

Example 2

*var* hobbies

=['reading','singing','cycling','sleeping','coding','travellin g','cricket'];

console.log(hobbies.includes('eating')); Output : false

Splice()- It helps to Modify an array.

In the splice() method,

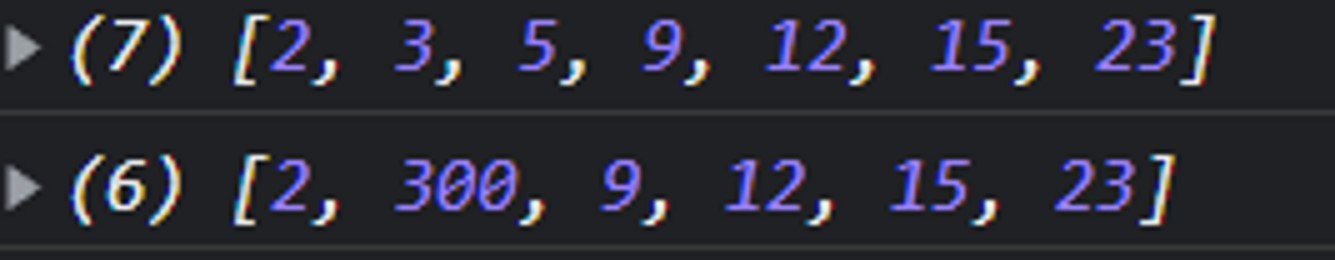
* The first argument is the index of an array to start removing an item from.
* The second argument is the number of elements that you want to remove from the index element.

Syntax: Splice(start index , delete count, items);

*var* num=[2,3,5,9,12,15,23];

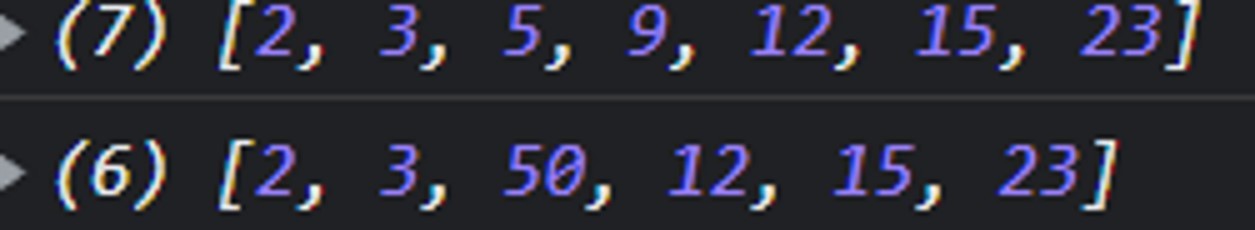
console.log(num); num.splice(1,2,300); console.log(num);

Output:



*var* num=[2,3,5,9,12,15,23];

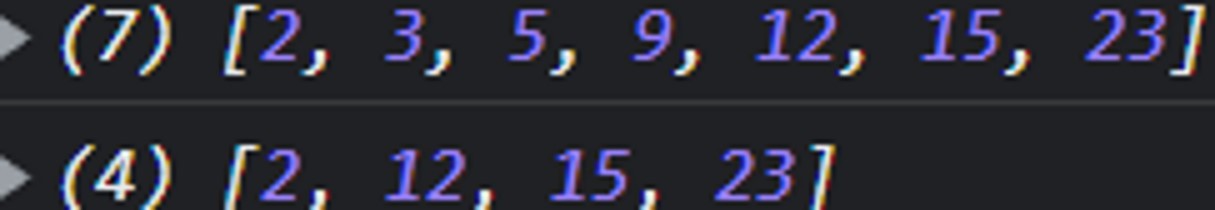
console.log(num); num.splice(2,2,50) console.log(num); Output:



*var* num=[2,3,5,9,12,15,23];

console.log(num); num.splice(1,3) console.log(num);

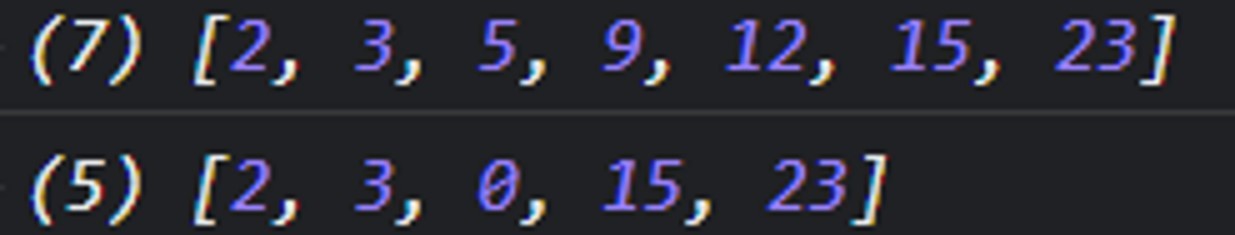
Output:



*var* num=[2,3,5,9,12,15,23];

console.log(num); num.splice(2,3,0) console.log(num);

Output:



slice()

It create new array instead of modifying in same array. It will not consider the last element.

Slice method consider start index and end index element will be ignore while slicing.

Syntax: Slice(start index , end index );

* + Difference between splice and slice

|  |  |
| --- | --- |
| splice | slice |
| Original will be modified | It create new array |
| Syntax: Splice(start index , delete count, items); | Syntax: Splice(start index , end index ); |
| Splice accept negative values | Slice will accept negative value and give empty string when we write -ve in starting index. |

[Variables](https://www.geeksforgeeks.org/javascript-variables/)

In JavaScript, variables are used to store and manage data. They are created using the var, let, or const keyword.

**var:**

Declares a variable. It has a function-scoped or globally-scoped behavior.

var x = 10;

### **let:**

Introduces block-scoped variables. It’s commonly used for variables that may change their value.

let y = "Hello";

### **const:**

Declares variables that cannot be reassigned. It’s block-scoped as well.

const PI = 3.14;

**Primitive Data Types**

The predefined data types provided by JavaScript language are known as primitive data types. Primitive data types are also known as in-built data types.

* [**Number:**](https://www.geeksforgeeks.org/javascript-numbers/) JavaScript numbers are always stored in double-precision 64-bit binary format IEEE 754. Unlike other programming languages, you don’t need int, float, etc to declare different numeric values.

1. [**String:**](https://www.geeksforgeeks.org/javascript-strings/) JavaScript Strings are similar to sentences. They are made up of a list of characters, which is essentially just an “array of characters, like “”/’’
2. [**Boolean:**](https://www.geeksforgeeks.org/javascript-boolean/) Represent a logical entity and can have two values: true or false.
3. [**Null:**](https://www.geeksforgeeks.org/null-in-javascript/) This type has only one value that is *null.*
4. [**Undefined:**](https://www.geeksforgeeks.org/undefined-in-javascript/) A variable that has not been assigned a value is *undefined.*
5. [**Symbol:**](https://www.geeksforgeeks.org/javascript-symbol-method/) Symbols return unique identifiers that can be used to add unique property keys to an object that won’t collide with keys of any other code that might add to the object.
6. **Syntax:**
7. [Symbol.peoperty](string)

The Symbol() function also accepts an optional object parameter. This object can be used to create a global Symbol, which can be accessed from anywhere in the code. A global Symbol can be created by passing in an object with a 'global' property set to true.

// Create a global Symbol

const myGlobalSymbol = Symbol('myGlobalSymbol', { global: true });

console.log(myGlobalSymbol);

// expected output: Symbol(myGlobalSymbol)

1. [**BigInt:**](https://www.geeksforgeeks.org/bigint-in-javascript/) BigInt is a built-in object in JavaScript that provides a way to represent whole numbers larger than 253-1.

BigInt is a built-in object in JavaScript that provides a way to represent whole numbers larger than 253-1. The largest number that JavaScript can reliably represent with the Number primitive is 253-1, which is represented by the **MAX\_SAFE\_INTEGER** constant. This has various uses where operations on large numbers are required.

BigInt is the second numeric data type in JavaScript (after Number).

let bigNum = BigInt(

"123422222222222222222222222222222222222");

console.log(bigNum);

// Parameter in hexadecimal format

let bigHex = BigInt("0x1ffffffeeeeeeeeef");

console.log(bigHex);

// Parameter in binary format

let bigBin = BigInt(

"0b1010101001010101001111111111111111");

console.log(bigBin);

<script>

let x = Number.MAX\_SAFE\_INTEGER;

document.getElementById("demo").innerHTML = x;

</script>

<script>

let x = Number.MIN\_SAFE\_INTEGER;

document.getElementById("demo").innerHTML = x;

</script>

Most programming languages have many number types:

Whole numbers (integers):  
byte (8-bit), short (16-bit), int (32-bit), long (64-bit)

Real numbers (floating-point):  
float (32-bit), double (64-bit).

**Javascript numbers are always one type:  
double (64-bit floating point).**

**How do you define a floating-point number?**

A floating-point number, is a positive or negative whole number with a decimal point. For example, 5.5, 0.25, and -103.342 are all floating-point numbers, while 91, and 0 are not.

**Non-Primitive Data Types**

The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.

* [**Object:**](https://www.geeksforgeeks.org/objects-in-javascript/) It is the most important data type and forms the building blocks for modern JavaScript. We will learn about these data types in detail in further articles.

**How to Define Functions in JavaScript**

## Function Declaration

The most basic setup for defining a function in JavaScript is through **function declaration**. You simply declare the function, give it a name, and include the parameters.

function some\_function(param1, param2) {

// Insert code here

}

A basic function might add the two parameters together and return the result.

let x = add\_params(2, 4); // x = 6

function add\_params(param1, param2) {

return param1 + param2;

}

**Return Keyword:**

JavaScript return statement is used to return a particular value from the function. The function will stop the execution when the return statement is called and return a specific value. The return statement should be the last statement of the function because the code after the return statement won't be accessible.

**Parameters:**

A parameter is one of the variables in a function. And when a method is called, the arguments are the data you pass into the method's parameters

## Function Expressions

A function in JavaScript can also be defined using a **function expression**. Function expressions allow the option of creating an anonymous function, i.e., a function without a name.

let some\_function = function(param1, param2) {

// Insert code here

};

We can add the parameters together in a similar way.

let add\_params = function(param1, param2) {

return param1 + param2;

};

let x = add\_params(2, 4); // x = 6

**HIGHER ORDER FUNCTION**

A “higher-order function” is a function that accepts functions as parameters and/or returns a function.

Function demo( ){

Clg(“hello this is demo starts”);

returns function( ){

clg(“HOF”)

}

}

demo( )( );

**CALLBACK FUNCTION**

A javascript callback function is function which is to be executed after another function has finished execution

A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

<script>

function per1(name,callback){

return function(){

console.log('my name is '+ name)

}

callback()

}

function per2(){

console.log('callback function')

}

let per=per1("manoj",per2())

per1()

console.log(per());

</script>

OBJECT

JavaScript Objects

A javaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.

We can can create an object by using 3 ways

1.By using literals

-Create a single object , using an object literals

Syntax: var obj={

color:"white",

width:123,

}

2.create an object by using new keyword

-Create a single object ,using new keyword.

Syntax: var obj1=new Object();

3.create an object by using constructor function

Syntax:

function person(name,age,sal){

this.name=name;

this.age=age;

this.sal=sal;

}

A function is a set of instructions that can be used repeatedly in a program. In JavaScript, functions are objects that can be declared, assigned, and passed as parameters to other functions.   
JavaScript functions come in different types and have various differences, especially in ES6 (ECMAScript 2015) and later versions. Here's a breakdown of JavaScript function types and some ES6 features related to functions:

**Function Types:**

1. **Named Functions:**

Named functions are function declarations that include a name. They are also sometimes referred to as traditional functions or function declarations.

function add(a, b) { return a + b; }

1. **Anonymous Functions:**

Anonymous functions are function expressions that do not have a name associated with them. They are also sometimes referred to as lambda functions or function expressions

const add = function(a, b) { return a + b; };

Anonymous functions are typically used when the same code needs to be executed only once or twice throughout a program. They can be assigned to variables or passed as parameters to other functions.

1. **Arrow Functions (Introduced in ES6):**

Arrow functions are a shorthand syntax for anonymous functions that was introduced in ECMAScript 6 (ES6). They are also sometimes referred to as fat arrow functions or lambda shorthand.

const add = (a, b) => a + b;

**Differences and Features:**

1. **Arrow Functions:**
   * Concise syntax: Arrow functions provide a shorter syntax compared to traditional function expressions.
   * Lexical **this**: Arrow functions do not have their own **this** value. Instead, **this** is lexically bound, meaning it's inherited from the enclosing scope.
   * Cannot be constructors: Arrow functions cannot be used as constructors with the **new** keyword.
   * No **arguments** binding: Arrow functions do not have their own **arguments** object. Instead, they inherit it from the enclosing scope.

**What is arguments in JavaScript ?**

The **arguments**is an object which is local to a function. You can think of it as a local variable that is available with all functions by default except arrow functions in JavaScript.

This object (arguments) is used to access the parameter passed to a function. It is only available within a function. We can’t access it outside the function. Arguments object allow you to access all of the arguments that are passed to a function. We can access these arguments using **indexes.**

**Return Keyword:**

The return statement stops the execution of a function and returns a value.

1. **Function Scoping:**
   * ES6 introduced **let** and **const** which have block scope, unlike **var** which has function scope. This affects how functions declared with these keywords are scoped.
2. **Default Parameters (Introduced in ES6):**
   * You can set default values for parameters directly in the function signature.

function greet(name = 'World') { console.log(`Hello, ${name}!`); } greet(); // Output: Hello, World! greet('John'); // Output: Hello, John!

1. **Rest Parameters (Introduced in ES6):**
   * Rest parameters allow a function to accept an indefinite number of arguments as an array. A rest operator is a type of parameter that gets all of the remaining parameters of a function call via an Array. It enables us to handle a variety of inputs as parameters in a function. Because it is used to combine many items, the rest operator is very helpful during array and object destructuring.
   * **rest syntax collects multiple elements and "condenses" them into a single element.**

function sum(...numbers) { return numbers.reduce((acc, curr) => acc + curr, 0); } console.log(sum(1, 2, 3, 4)); // Output: 10

1. **Spread Syntax (Introduced in ES6):**
   * The spread syntax allows an iterable such as an array expression or string to be expanded in places where zero or more arguments or elements are expected.
   * **Spread syntax "expands" an array into its elements**

const arr1 = [1, 2, 3]; const arr2 = [4, 5, 6]; const combinedArray = [...arr1, ...arr2]; // Output: [1, 2, 3, 4, 5, 6]

1. **Template Literals (Introduced in ES6):**
   * Template literals provide an easy way to interpolate variables into strings using **${}** syntax.

const name = 'John'; console.log(`Hello, ${name}!`); // Output: Hello, John!

These features introduced in ES6 enhance the capabilities and expressiveness of JavaScript functions, making them more concise and powerful.

JavaScript Objects

A javaScript object is an entity having state and behavior (properties and method). For example: car, pen, bike, chair, glass, keyboard, monitor etc.

JavaScript is an object-based language. Everything is an object in JavaScript.

JavaScript is template based not class based. Here, we don't create class to get the object. But, we direct create objects.

## Creating Objects in JavaScript

There are 3 ways to create objects.

1. By object literal
2. By creating instance of Object directly (using new keyword)
3. By using an object constructor (using new keyword)

## 1) JavaScript Object by object literal

The syntax of creating object using object literal is given below:

1. object={property1:value1,property2:value2.....propertyN:valueN}

As you can see, property and value is separated by : (colon).

Let’s see the simple example of creating object in JavaScript.

**<script>**

let emp={id:102,name:"Shyam Kumar",salary:40000}

console.log(emp.id+" "+emp.name+" "+emp.salary);

**</script>**

#### **Output of the above example**

102 Shyam Kumar 40000

## 2) By creating instance of Object

The syntax of creating object directly is given below:

1. var objectname=new Object( );

Here, **new keyword** is used to create object.

Let’s see the example of creating object directly.

**<script>**

var emp=new Object();

emp.id=101;

emp.name="Ravi Malik";

emp.salary=50000;

console.log(emp.id+" "+emp.name+" "+emp.salary);

**</script>**

#### **Output of the above example**

101 Ravi 50000

## 3) By using an Object constructor

Here, you need to create function with arguments. Each argument value can be assigned in the current object by using this keyword

The **this keyword** refers to the current object.

The example of creating object by object constructor is given below.

**<script>**

function emp(id,name,salary){

this.id=id;

this.name=name;

this.salary=salary;

}

let e=new emp(103,"Vimal Jaiswal",30000);

Console.log(e.id+" "+e.name+" "+e.salary);

**</script>**

#### **Output of the above example**

103 Vimal Jaiswal 30000

**OBJECT METHODS**

**REFERE CLASS ROOM CODE FOR OBJECT METHODS.**

//Date object

-Js date object represent a single moment in time in a platform

independent format.

-date object contains numbers that represents miliseconds since 1 january

1970

Date object methods

//date object

// console.log("date object for regular date");

var a=new Date();

// console.log(a);

//date (year,mon,day,hr,min,sec) 0=jan ,feb=1

// console.log("manual date");

// var b=new Date(2022,11,5,2,34,33);

// console.log(b);

// // //1sec=1000milisec

// var c=new Date(1000);

// console.log(c);

// //date (date should be in string formate manual input)

// var d=new Date("25,jan,2022");

// console.log(d);

// //get the date

console.log(a.getDate());

console.log(a.getMonth()+1);

console.log(a.getFullYear());

console.log(a.getTime());

console.log(a.getHours());

console.log(a.getDay());

// //25/12/2022

console.log(a.getDate()+"/"+a.getMonth()+"/"+a.getFullYear());

console.log("27"+"/"+"jan"+"/"+"2022");

//Math objects

-Math is inbuilt object that has properties and methods for mathematical

constant and functions.its not a function object

-math is work with number type

math object methods

var a= Math.PI;

console.log(a);

document.write(a);

//round

var b=Math.round(12.74);//round the nearest int value

console.log(b);

//power

var c= Math.pow(2,3);

console.log(c);

//absolute it can conver -value to positive

var d=Math.abs(-12);

console.log(d);

//Square root

var e=Math.sqrt(16);

console.log(e);

//floor give 12.49 or 12.99 it will give the floor value 12

var f=Math.floor(12.99);

console.log(f);

//ceil give 12.49 or 12.99 it will give the next vaule (ceil)

var g=Math.ceil(12.44);

console.log(g);

//max

var h=Math.max(22,33,44,55,66,77,88,11);

console.log(h);

//min

var i=Math.min(22,33,44,55,66,77,88,11);

console.log(i);

//random

var j= Math.random();

console.log(j);

**Looping**

It is also called iteration.

The process of executing a instruction /block of instruction repeatedly multiple times.

Note: when we design a loop it is a responsibility of a programmer to break the loop after achieving the desire task, if not the

program get into infinite loop statement.

If we don’t break loop it will be infinite and control will not come out of loop.

Condition is use to break loop after some certain

Loop statements

1.While

a. Syntax

while(condition){

statement to be repeated;

}

2.Do-while :

The do-while loop loops through a

The do-while loop loops through a block of code once, then the condition is evaluated. If the condition is

true the statement is repeated as long as specified condition is true.

a. number of iteration in do while is 1

Syntax

do{

// statement;

}

while(condition);

**Note:**

a. in do while loop the body of the loop is executed first then the condition is evaluated.

b. If the condition evaluated is true the body of the loop inside the do statement is executed again.

c. The statement inside the do statement is executed again.

d. This process continue until the condition evaluates to false. Then the loop stops.

3. For

4. For-in etc

**Functions**

• Function is a block of instruction which is used to perform a specific task

• A function get executed only when it called.

• The main advantage of function is we can achieve code reusability.

• To call a function we need its reference and ()

**Note**: in javascript functions are beautiful, every function is nothing but an object.

Syntax to create a function.

Generally we can create a function in 5 ways:

1. Function declaration statement(function statement).

2. Function expression.

Function declaration / statement(function statement).

Syntax :

function identifier ([list\_of\_parameter,...])

{

statements;

}

1.

Note:

i. Function is object

ii. Name of function is variable which holds the reference of function object.

iii. Creating a function using function statement supports function hoisting.

iv. Therefore we can also call a function before function declaration.

v. when we try to log function name the entire function defination is printed.

console.log('start');

console.log(test);

function test(){

console.log('Hello');

}

console.log('start');

test - it is a function variable - it will return body of function

test () - invoking a function.

To call a function - function\_name();// we can also pass data

Function\_name(arguments\_list,.....);

console.log('start');

function test(){

console.log('Hello');

}

test();

test();

test();

console.log('start');

parameter(placeholder: variable to hold data when function is called)

console.log('start');

function test(a){

console.log('Hello');

console.log(a);

}

test(10); argument : values which are passed.

console.log('start');

Parameters

The variables declared in the function defination is known as parameters.

The parameters have local scope (can be used only inside function body).

Parameters are used to hold the values passed by calling a function.

Eg:

function sum(a, b){

console.log(a + b)

}

// a and b are variables local to the function sum

Arguments

The values passed in the method call statement is known as arguments.

Note : An argument can be a literal, variable or an expression which gives a results.

Eg 1:

sum(10,20); // 10,20 are literals used as arguments

Eg 2:

sum(-10+3,-20);//-27

Eg 3:

sum(a ,b); //a and b are variable used as argument

**return**

1. It is a keyword used as control transfer statement in a function.

Return will stop the execution of the function and transfer control along with data

to the caller.

2.

function toMeters(cms){

return (cms/100);

}

var cms =2546;

console.log(toMeters(cms));

var m =toMeters(cms );

console.log(m);

The ability of a function to use as value is called first class function

**2. Function as Expression**

Syntax :

var/let/const identifier = function (){

}

The function is used as value

Disadvantage :

• function is not hoisted

• We cannot use a function before declaration.

a();// error

var a = function (){ /\* a has address of function\*/

console.log("fun");// will get entire function definition

}

In the above example function is not hoisted instead variable is

hoisted and assigned with default value undefined. therefore type of

a is not function it is undefined.

**This**

It is a keyword used as variable.

It holds the address of global window object.

Therefore with the help of this variable we can use members of global window object.

Whenever we call a function a new execution context is created. Inside that all local

variable declare inside will be there and 1 more "this" will be there which is different from

this

Inside this it have address of which function belongs to.

a.In javascript 'this' is a property of every function.(every function will have this

keyword)

b. Generally this contains the current execution context to which the function belongs.

**3.Arrow Functions**

• Arrow function was introduce from ES6 OF JS.

• The main function of using arrow function is to reduce syntax

Syntax

( parameter\_list,...) => {}

Note:

1. Parameter is optional.

2. If function has only one statement , then block is optional.

3.it is mandatory to create a block if return keyword is used.

Error if we write like this

const c =(n1+n2) => return n1+n2;

console.log(c(10,20));

Correct solution

const c =(n1, n2) => { return n1+n2};

console.log(c(10,20));

Operation taking parameter using arrow function

**4.IIFE-(Immediate invocation function)**

When a function is called immediately as soon as the function object is created.

Steps to achieve

• Treat a function like a expression by declaring in pair of bracket.

Add another pair pf braces next to it which behaves like function call statement.

○ Eg1

(function abc(){

console.log("Hi");

})();

○ Eg2

let a= ( ()=>{console.log("hello"); return 10})();

console.log(a);

**5.Anonymous function**

The function declare without any name which is called as anonymous function

Syntax:

function (){

instruction

}

**//OBJECTS**

-js objects is designed on simple object based paradigm.An object is a

collection of properties, and a properties is association between a

name(or key) and a value.

We can can create an object by using 3 ways

1.By using literals

-Create a single object , using an object literals

Syntax: var obj={

color:"white",

width:123,

}

2.create an object by using new keyword

-Create a single object ,using new keyword.

Syntax: var obj1=new Object();

3.create an object by using constructor function

Syntax:

function person(name,age,sal){

this.name=name;

this.age=age;

this.sal=sal;

}

//Date object

-Js date object represent a single moment in time in a platform

independent format.

-date object contains numbers that represents miliseconds since 1 january

1970

Date object methods

//date object

// console.log("date object for regular date");

var a=new Date();

// console.log(a);

//date (year,mon,day,hr,min,sec) 0=jan ,feb=1

// console.log("manual date");

// var b=new Date(2022,11,5,2,34,33);

// console.log(b);

// // //1sec=1000milisec

// var c=new Date(1000);

// console.log(c);

// //date (date should be in string formate manual input)

// var d=new Date("25,jan,2022");

// console.log(d);

// //get the date

console.log(a.getDate());

console.log(a.getMonth()+1);

console.log(a.getFullYear());

console.log(a.getTime());

console.log(a.getHours());

console.log(a.getDay());

// //25/12/2022

console.log(a.getDate()+"/"+a.getMonth()+"/"+a.getFullYear());

console.log("27"+"/"+"jan"+"/"+"2022");

//Math objects

1310868812

-Math is inbuilt object that has properties and methods for mathematical

constant and functions.its not a function object

-math is work with number type

math object methods

var a= Math.PI;

console.log(a);

document.write(a);

//round

var b=Math.round(12.74);//round the nearest int value

console.log(b);

//power

var c= Math.pow(2,3);

console.log(c);

//absolute it can conver -value to positive

var d=Math.abs(-12);

console.log(d);

//Square root

var e=Math.sqrt(16);

console.log(e);

//floor give 12.49 or 12.99 it will give the floor value 12

var f=Math.floor(12.99);

console.log(f);

//ceil give 12.49 or 12.99 it will give the next vaule (ceil)

var g=Math.ceil(12.44);

console.log(g);

//max

var h=Math.max(22,33,44,55,66,77,88,11);

console.log(h);

//min

var i=Math.min(22,33,44,55,66,77,88,11);

console.log(i);

//random

var j= Math.random();

console.log(j);

JavaScript Date Object

The **JavaScript date** object can be used to get year, month and day. You can display a timer on the webpage by the help of JavaScript date object.

You can use different Date constructors to create date object. It provides methods to get and set day, month, year, hour, minute and seconds.

## Constructor

You can use 4 variant of Date constructor to create date object.

1. Date()
2. Date(milliseconds)
3. Date(dateString)
4. Date(year, month, day, hours, minutes, seconds, milliseconds)

## JavaScript Date Methods



# JavaScript Math

The **JavaScript math** object provides several constants and methods to perform mathematical operation. Unlike date object, it doesn't have constructors.

Synatax

- Math.pow(base value,power)

- Math.sqrt(value)

- Math.max(v1,v2,v3,vn)

- Math.min(v1,v2,v3,vn)

- Math.abs(negative vaue)

- Math.floor(value)

- Math.ceil(value)

- Math.random( )

- Math.round(value)

- Math.PI( )

**OBJECT DESTRUCTURING**

JavaScript Object Destructuring is an expression which allows us to access the data from objects like arrays, objects, maps and assign it to new variables. Through this object destructuring, we can create variables easily from the object's properties

Syntax:

Variable{key1:variable1,key2:variable2,keyn:variablen}

Ex-

let obj ={

name:`ganesh`,

age:25,

status:true,

}

Console.log(obj);

Destructuring

let {name:objName,age:objAge,status:objSatus}=obj;

**ARRAY DESTRUCTURING**

Destructuring means to break down a complex structure into simpler parts. With the syntax of destructuring, you can extract smaller fragments from objects and arrays. It can be used for assignments and declaration of a variable.

Syntax-

let [variable1,variable2,variablen]=array refVariable;

let array=[10,20,30,40]

console.log(array)

Destructuring

let[value1,value2,value3,value4]=array

console.log(value1+value2+value3+value4)//100

# JavaScript Array map() method

The JavaScript array map() method calls the specified function for every array element and returns the new array. This method doesn't change the original array.

## Syntax

The map() method is represented by the following syntax:

1. array.map(callback(currentvalue,index,arr),thisArg)

## Parameter

**callback** - It represents the function that produces the new array.

**currentvalue** - The current element of array.

**index** - It is optional. The index of current element.

**arr** - It is optional. The array on which map() method operated.

**thisArg** - It is optional. The value to use as this while executing callback.

## Return

A new array whose each element generate from the result of a callback function.

## JavaScript Array map() method example

let fName=[“ajay”,”atul”,”jay”];

let fullNAme=fName.map((cValue)=>{

return cValue+ “ patil”

})

Console.log(fullName);

[“ajay patil ”,”atul patil”, “ jay patil” ]

# JavaScript Array filter() method

The JavaScript array filter() method filter and extract the element of an array that satisfying the provided condition. It doesn't change the original array.

## Syntax

The filter() method is represented by the following syntax:

1. array.filter(callback(currentvalue,index,arr),thisArg)

## Parameter

**callback** - It represents the function that test the condition.

**currentvalue** - The current element of array.

**index** - It is optional. The index of current element.

**arr** - It is optional. The array on which filter() operated.

**thisArg** - It is optional. The value to use as this while executing callback.

## Return

A new array containing the filtered elements.

## JavaScript Array filter() method example

let array=[10,20,30,40,50,60,70]

let arrfil= array.filter((cValue)=>{

if(cValue>10 && cValue<60){

return cValue;

}

})

Console.log(arrfil)// [20,30,40,50]

In [Javascript,](https://www.geeksforgeeks.org/javascript-tutorial/)both the [spread operator](https://www.geeksforgeeks.org/javascript-spread-operator/) and [rest parameter](https://www.geeksforgeeks.org/javascript-rest-operator/) have the same syntax which is three dots**(…)**. Even though they have the same syntax they differ in functions.

[**Spread operator:**](https://www.geeksforgeeks.org/javascript-spread-operator/)The [spread](https://www.geeksforgeeks.org/javascript-spread-operator/) operator helps us expand an iterable such as an array where multiple arguments are needed, it also helps to expand the object expressions. In cases where we require all the elements of an iterable or object to help us achieve a task, we use a spread operator.

**Syntax:**

Var name = [...it7erable];

<script>

var array1 = [10, 20, 30, 40, 50];

var array2 = [60, 70, 80, 90, 100];

var array3 = [...array1, ...array2];

console.log(array3);

</script>

[**Rest operator:**](https://www.geeksforgeeks.org/javascript-rest-operator/)The [rest](https://www.geeksforgeeks.org/javascript-rest-operator/)parameter is converse to the spread operator. while the spread operator expands elements of an iterable, the rest operator compresses them. It collects several elements. In functions when we require to pass arguments but were not sure how many we have to pass, the rest parameter makes it easier.

**Syntax:**

function function\_name(...arguments) {

statements;

}

Asynchronous is a non-blocking architecture, so the execution of one task isn't dependent on another. Tasks can run simultaneously. Synchronous is a blocking architecture, so the execution of each operation depends on completing the one before it. Each task requires an answer before moving on to the next iteration.

The differences between asynchronous and synchronous include:

* **Async** is multi-thread, which means operations or programs can run in parallel.
* **Sync** is a single-thread, so only one operation or program will run at a time.
* **Async** is non-blocking, which means it will send multiple requests to a server.
* **Sync** is blocking — it will only send the server one request at a time and wait for that request to be answered by the server.
* **Async** increases throughput because multiple operations can run at the same time.
* **Sync** is slower and more methodical.

Promises:

In JavaScript, a Promise is an object that will produce a single value some time in the future. If the promise is successful, it will produce a resolved value, but if something goes wrong then it will produce a reason why the promise failed. The possible outcomes here are similar to that of promises in real life.

JavaScript promises can be in one of three possible states.

These states indicate the progress of the promise. They are:

* *pending*: This is the default state of a defined promise
* *fulfilled*:  This is the state of a successful promise
* *rejected*: This is the state of a failed promise

**How to Create a Promise in JavaScript**

To create a promise, you need to create an instance object using the Promise constructor function. The Promise constructor function takes in one parameter. That parameter is a function that defines when to resolve the new promise, and optionally when to reject it.

const promise = new Promise((resolve, reject) => {

// Condition to resolve or reject the promise

});

Ex. const promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("Done!"), 2000);

});

**Async/Await Function in JavaScript**

**Async**:

It simply allows us to write promises-based code as if it was synchronous and it checks that we are not breaking the execution thread. It operates asynchronously via the event loop. Async functions will always return a promise. It makes sure that a promise is returned and if it is not returned then JavaScript automatically wraps it in a promise which is resolved with its value.

**Await:**

Await function is used to wait for the promise. It could be used within the async block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

let prom1=new Promise((resolve,reject)=>{

setTimeOut(( )=>{

let pro=”I promise you ill never leave you”

resolve(pro)

},3000)

})

let prom2=new Promise((resolve,reject)=>{

setTimeOut(( )=>{

let pro=”I will gift you iphone on your birthday”

resolve(pro)

},6000)

})

Async function getPromise( ){

let pro1=await prom1;

console.log(pro1);

let pro2= await prom2;

console.log(pro2)

}

getPromise( )

what is dom

DOM stands for Document Object Model. It is a programming interface for web documents, representing the structure and content of an HTML, XML, or XHTML document as a tree-like model. The DOM allows scripts or programming languages, such as JavaScript, to access and manipulate the elements and properties of a web page.

targeting methods

getElementById(id) - Returns the element with the specified ID.

getElementsByClassName(class) - Returns a collection of elements with the specified class name.

getElementsByTagName(tag) - Returns a collection of elements with the specified tag name.

querySelector(selector) - Returns the first element that matches the specified CSS selector.

querySelectorAll(selector) - Returns a list of all elements that match the specified CSS selector

classListmethods

classlist methods

The classList property is a read-only property that returns a live DOMTokenList collection of the class attributes of an element.

.add()- Adds one or more class names to the element. If the class name already exists, it will not be added again.

remove: Removes one or more CSS classes from the element.

toggle: Toggles the presence of a CSS class on the element. If the class is already present, it removes it; otherwise, it adds it.

contains: Checks if the element has a specific CSS class. It returns a boolean value indicating the presence or absence of the class.

DOM Travising method

DOM traversing refers to the process of navigating and accessing different elements within the Document Object Model (DOM) tree structure using JavaScript. It allows you to find specific elements, move between parent and child elements, or locate sibling elements.

parentNode: This property returns the parent node of the specified element/node.

parentElement: This property returns the parent element of the specified element/node

Child Nodes: The childNodes property is used to access a collection of all child nodes of an element, including text nodes, element nodes, and other types of nodes.

Children: The children property is similar to childNodes, but it specifically returns only the child elements (element nodes) of an element, excluding text nodes or other non-element nodes.

nextSibling and previousSibling: These properties return the next or previous sibling node of the specified element.

firstChild and lastChild: These properties return the first or last child node of the specified element.

dom mouse events

click: The click event occurs when the user clicks the mouse button on an element.

dblclick: The dblclick event occurs when the user double-clicks the mouse button on an element.

mousedown: The mousedown event occurs when the user presses the mouse button down on an element.

mouseenter and mouseleave: The mouseenter event occurs when the mouse pointer enters an element, while the mouseleave event occurs when the mouse pointer leaves the element

mousemove: The mousemove event occurs when the mouse pointer is moved over an element.

dom addEventListener

addEventListener is a JavaScript method used to register a function to be called in response to a specific event that occurs on a DOM element.

ex- document.getElementById().addEventListener("event",function(){})

form events

onsubmit: This event is triggered when a form is submitted. You can use this event to perform form validation or submit the form data to a server.

onreset: This event is triggered when the reset button of a form is clicked. You can use this event to reset the form data to its initial values.

onchange: This event is triggered when the value of a form element is changed. You can use this event to update other parts of the webpage based on the user's input.

onfocus: This event is triggered when a form element receives focus. You can use this event to provide visual feedback or perform other actions when the user interacts with the element.

onblur: This event is triggered when a form element loses focus. You can use this event to perform form validation or other actions when the user finishes interacting with the element.

onkeydown, onkeyup, and onkeypress: These events are triggered when a key is pressed, released, or pressed and released, respectively. You can use these events to perform actions based on the user's keyboard input.

create method

createElement: This method creates a new HTML element with the specified tag name.

createTextNode: This method creates a new text node with the specified text content.'

append method

the appendChild() method is used to add a new child node to an existing parent node in the Document Object Model (DOM).

insertbefore method - it used to set the position in div using children or childnodes

insert adjacent element-This method allows you to insert a new element as a sibling before or after the reference element.

get and set attribute

getAttribute is used to get values from that object name.

setAttribute() sets the data into the database and getAttribute() fetch the data from the database.

replace and removechild

replaceChild: This method replaces a child element with a new element within its parent.

removeChild: This method removes a child element from its parent.

set intervalmethod

the setInterval function is used to repeatedly execute a function or code snippet at a specified interval. It allows you to schedule a function to run repeatedly with a fixed time delay between each execution.

set timeout

the setTimeout() method is used to execute a function or evaluate an expression after a specified delay (in milliseconds). It allows you to schedule a single execution of a function or code block after a certain amount of time has passed.

ES6 Spread Operator

[ES6](https://www.javatpoint.com/es6) introduced a new operator referred to as a spread operator, which consists of three dots (...). It allows an iterable to expand in places where more than zero arguments are expected. It gives us the privilege to obtain the parameters from an array.

Spread operator syntax is similar to the rest parameter, but it is entirely opposite of it. Let's understand the syntax of the spread operator.

### **Syntax**

1. var variablename1 = [...value];

The three dots (...) in the above syntax are the spread operator, which targets the entire values in the particular variable.

## Spread Operator and Array Manipulation

Here, we are going to see how we can manipulate an array by using the spread operator.

### **Constructing array literal**

When we construct an array using the literal form, the spread operator allows us to insert another array within an initialized array.

**Example**

let colors = ['Red', 'Yellow'];

let newColors = [...colors, 'Violet', 'Orange', 'Green'];

console.log(newColors);

**Output**

[ 'Red', 'Yellow', 'Violet', 'Orange', 'Green' ]

### **Concatenating arrays**

Spread operator can also be used to concatenate two or more arrays.

**Example**

1. let colors = ['Red', 'Yellow'];
2. let newColors = [...colors, 'Violet', 'Orange', 'Green'];
3. console.log(newColors);

**Output**

[ 'Red', 'Yellow', 'Violet', 'Orange', 'Green' ]

### **Copying an array**

We can also copy the instance of an array by using the spread operator.

**Example**

let colors = ['Red', 'Yellow'];

let newColors = [...colors];

console.log(newColors);

**Output**

[ 'Red', 'Yellow' ]

If we copy the array elements without using the spread operator, then inserting a new element to the copied array will affect the original array.

But if we are copying the array by using the spread operator, then inserting an element in the copied array will not affect the original array.

**Example**

**Without using spread operator**

let colors = ['Red', 'Yellow'];

let newColors = colors;

newColors.push('Green');

console.log(newColors);

console.log(colors);

**Output**

[ 'Red', 'Yellow', 'Green' ]

[ 'Red', 'Yellow', 'Green' ]

**Using spread operator**

let colors = ['Red', 'Yellow'];

let newColors = [...colors];

newColors.push('Green');

console.log(newColors);

console.log(colors);

**Output**

[ 'Red', 'Yellow', 'Green' ]

[ 'Red', 'Yellow' ]

#### **Note: Instead of elements, the spread operator only copies the array itself to the new one. It means that the operator can do a shallow copy instead of a deep copy.**

## Spread operator and Strings

Let's see how the spread operator spreads the strings. The illustration for the same is given below.

**Example**

Here, we have constructed an array **str** from individual strings.

1. let str = ['A', ...'EIO', 'U'];
2. console.log(str);

In the above example, we have applied the spread operator to the string **'EIO'**. It spreads out each specific character of the **'EIO'** string into individual characters.

We will get the following output after the execution of the above code.

**Output**

[ 'A', 'E', 'I', 'O', 'U' ]

# JavaScript Promise

Promises in real-life express a trust between two or more persons and an assurance that a particular thing will surely happen. In javascript, a Promise is an object which ensures to produce a single value in the future (when required). Promise in javascript is used for managing and tackling asynchronous operations.

## Need for JavaScript Promise

Till now, we learned about events and callback functions for handling the data. But, its scope is limited. It is because events were not able to manage and operate asynchronous operations. Thus, Promise is the simplest and better approach for handling asynchronous operations efficiently.

There are two possible differences between Promise and Event Handlers:

1. A Promise can never fail or succeed twice or more. This can happen only once.
2. A Promise can neither switch from success to failure, or failure to success. If a Promise has either succeeded or failed, and after sometime, if any success/failure callback is added, the correct callback will be invoked, no matter the event happened earlier.

## Constructor in Promise

|  |  |
| --- | --- |
| new Promise(function(resolve, reject){}); | Here, resolve(thenable) denotes that the promise will be resolved with then(). Resolve(obj) denotes promise will be fulfilled with the object Reject(obj) denotes promise rejected with the object. |

## Terminology of Promise

A promise can be present in one of the following states:

1. **pending:** The pending promise is neither rejected nor fulfilled yet.
2. **fulfilled:** The related promise action is fulfilled successfully.
3. **rejected:** The related promise action is failed to be fulfilled.
4. **settled:** Either the action is fulfilled or rejected.

Thus, a promise represents the completion of an asynchronous operation with its result. It can be either successful completion of the promise, or its failure, but eventually completed. Promise uses a **then()** which is executed only after the completion of the promise resolve.

## Promises of Promise

A JavaScript Promise promises that:

1. Unless the current execution of the js event loop is not completed (success or failure), callbacks will never be called before it.
2. Even if the callbacks with then() are present, but they will be called only after the execution of the asynchronous operations completely.
3. When multiple callbacks can be included by invoking then() many times, each of them will be executed in a chain, i.e., one after the other, following the sequence in which they were inserted.

let pro=new Promise((resolve,reject)=>{

setTimeout(()=>{

let shadi="i promise u ill never leave you!!!!"

//    resolve(shadi)

reject("gharvale nhi manenge")

},2000)

})

pro.then((shadiFinal)=>{

console.log(shadiFinal);

}).catch((error)=>{

console.log("gharvale nhi manenge");

})

1. **Resolve:** When the promise is executed successfully, the resolve argument is invoked, which provides the result.
2. **Reject:** When the promise is rejected, the reject argument is invoked, which results in an error.

It means either resolve is called or reject is called. Here, then() has taken one argument which will execute, if the promise is resolved. Otherwise, catch() will be called with the rejection of the promise.

What is ES6?

Understanding these Features

1. let and const keywords :

2. Arrow Functions

3. Multi-line Strings

4. Default Parameters

5. Template Literals

6. Destructuring Assignment

7. Enhanced Object Literals

8. Promises

9. Classes

10. Modules

1. let and const keywords :

The keyword "let" enables the users to define variables and on the other hand, "const" enables the users to define constants. Variables were previously declared using "var" which had function scope and were hoisted to the top. It means that a variable can be used before declaration. But, the "let" variables and constants have block scope which is surrounded by curly-braces "{}" and cannot be used before declaration.

let i = 10;

console.log(i); //Output 10

const PI = 3.14;

console.log(PI); //Output 3.14

2. Arrow Functions

ES6 provides a feature known as Arrow Functions. It provides a more concise syntax for writing function expressions by removing the "function" and "return" keywords.

Arrow functions are defined using the fat arrow (=>) notation.

// Arrow function

let sumOfTwoNumbers = (a, b) => a + b;

console.log(sum(10, 20)); // Output 30

It is evident that there is no "return" or "function" keyword in the arrow function declaration.

We can also skip the parenthesis in the case when there is exactly one parameter, but will always need to use it when you have zero or more than one parameter.

But, if there are multiple expressions in the function body, then we need to wrap it with curly braces ("{}"). We also need to use the "return" statement to return the required value.

3. Multi-line Strings

ES6 also provides Multi-line Strings. Users can create multi-line strings by using back-ticks(`).

It can be done as shown below :

let greeting = `Hello World,

Greetings to all,

Keep Learning and Practicing!`

4. Default Parameters

In ES6, users can provide the default values right in the signature of the functions. But, in ES5, OR operator had to be used.

//ES6

let calculateArea = function(height = 100, width = 50) {

// logic

}

//ES5

var calculateArea = function(height, width) {

height = height || 50;

width = width || 80;

// logic

}

5. Template Literals

ES6 introduces very simple string templates along with placeholders for the variables. The syntax for using the string template is ${PARAMETER} and is used inside of the back-ticked string.

let name = `My name is ${firstName} ${lastName}`

6. Destructuring Assignment

Destructuring is one of the most popular features of ES6. The destructuring assignment is an expression that makes it easy to extract values from arrays, or properties from objects, into distinct variables.

There are two types of destructuring assignment expressions, namely, Array Destructuring and Object Destructuring. It can be used in the following manner :

//Array Destructuring

let fruits = ["Apple", "Banana"];

let [a, b] = fruits; // Array destructuring assignment

console.log(a, b);

//Object Destructuring

let person = {name: "Peter", age: 28};

let {name, age} = person; // Object destructuring assignment

console.log(name, age);

7. Enhanced Object Literals

ES6 provides enhanced object literals which make it easy to quickly create objects with properties inside the curly braces.

function getMobile(manufacturer, model, year) {

return {

manufacturer,

model,

year

}

}

getMobile("Samsung", "Galaxy", "2020");

8. Promises

In ES6, Promises are used for asynchronous execution. We can use promise with the arrow function as demonstrated below.

var asyncCall = new Promise((resolve, reject) => {

// do something

resolve();

}).then(()=> {

console.log('DON!');

})

9. Classes

Previously, classes never existed in JavaScript. Classes are introduced in ES6 which looks similar to classes in other object-oriented languages, such as C++, Java, PHP, etc. But, they do not work exactly the same way. ES6 classes make it simpler to create objects, implement inheritance by using the "extends" keyword and also reuse the code efficiently.

In ES6, we can declare a class using the new "class" keyword followed by the name of the class.

class UserProfile {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

getName() {

console.log(`The Full-Name is ${this.firstName} ${this.lastName}`);

}

}

let obj = new UserProfile('John', 'Smith');

obj.getName(); // output: The Full-Name is John Smith

10. Modules

Previously, there was no native support for modules in JavaScript. ES6 introduced a new feature called modules, in which each module is represented by a separate ".js" file. We can use the "import" or "export" statement in a module to import or export variables, functions, classes or any other component from/to different files and modules.

export var num = 50;

export function getName(fullName) {

//data

};

import {num, getName} from 'module';

console.log(num); // 50

**ARRAY REDUCE( )**

The Javascript arr.reduce() method in JavaScript is used to reduce the array to a single value and executes a provided function for each value of the array (from left to right) and the return value of the function is stored in an accumulator.

Syntax:

array.reduce( function(total, currentValue, currentIndex, arr),

initialValue )

Parameters: This method accepts five parameters as mentioned above and described below:

function(total, currentValue, index, arr): It is the required parameter and is used to run for each element of the array. It contains four parameters which are listed below:

total: It is a required parameter and used to specify the initialValue or the previously returned value of the function.

currentValue: It is a required parameter and is used to specify the value of the current element.

currentIndex: It is an optional parameter and is used to specify the array index of the current element.

arr: It is an optional parameter and is used to specify the array object the current element belongs to.

initialValue: It is an optional parameter and is used to specify the value to be passed to the function as the initial value.

This example uses reduce() method to return the sum of all array elements.

let arr = [10, 20, 30, 40, 50, 60];

function sumofArray(sum, num) {

return sum + num;

}

function demo(item) {

console.log(arr.reduce(sumofArray));

}

demo()

JavaScript setTimeout() method

The **setTimeout()** method in JavaScript is used to execute a function after waiting for the specified time interval. This method returns a numeric value that represents the ID value of the timer.

Syntaxt

1. setTimeout(function, milliseconds);

ex function demo( ) {

console.log(“hello”)

}

setTimeout(demo,2000)

JavaScript setInterval() method

The setInterval() method in JavaScript is used to repeat a specified function at every given time-interval. It evaluates an expression or calls a function at given intervals. This method continues the calling of function until the window is closed or the clearInterval() method is called

## How to stop the execution?

We can use the **clearInterval()** method to stop the execution of the function specified in **setInterval()** method. The value returned by the **setInterval()** method can be used as the argument of **clearInterval()** method to cancel the timeout.

Syntaxt

1. setInterval(function, milliseconds);

let ele=document.getElementById("print")

// let num=0;

// let timer;

//   function setTimeInterval() {

//        ele.innerHTML="loading...."

//        timer=setInterval((welcome)=>{

//            ele.innerHTML=`i m guessing your lucky number  is it ${num}`

//            ++num;

//        },1000)

//   }

//   function stopTimer() {

//     clearInterval(timer)

//   }

# ES6 Rest Parameter

The rest parameter is introduced in ECMAScript 2015 or [ES6](https://www.javatpoint.com/es6), which improves the ability to handle parameters. The rest parameter allows us to represent an indefinite number of arguments as an array. By using the rest parameter, a function can be called with any number of arguments.

Before ES6, the **arguments** object of the function was used. The **arguments** object is not an instance of the Array type. Therefore, we can't use the **filter()** method directly.

The rest parameter is prefixed with three dots (...). Although the syntax of the rest parameter is similar to the spread operator, it is entirely opposite from the spread operator. The rest parameter has to be the last argument because it is used to collect all of the remaining elements into an array.

Syntax …variable

function show(...args) {

let sum = 0;

**for** (let i of args) {

sum += i;

}

console.log("Sum = "+sum);

}

show(10, 20, 30);

**JAVASCRIPT**

Javascript was invented by Brendan Eich on

September 1995.

Initially it was named as mocha,then named as

live script and now its named as Java script.

Javascript is a High-level,object-oriented

programming language.

Javascript is a popular programming language

used for adding dynamic nature to the websites

and it is also used to create web applications.

THERE IS NOTHING YOU CANT DO WITH

JAVASCRIPT(ALMOST)

Angular,React and vue are popular javascript

libraries or frameworks used for creating frontend

applications (or) user interfaces.

React native uses javascript to create mobile

applications.

Electron uses javascript to create desktop

applications.

Javascript basically runs on the browser.

Browsers have Javascript Engine to run the

javascript code on the browser.

We can use javascript outside the browser by

using node.js.

Developers console is basically used to write

javascript snippets.Of course,this is not used for

creating real time javascript applications.

How to open developers console?

Right click on chrome->inspect->console

we can link javascript file to our HTML file using

script tag by taking the help of src attribute!!!

javascript files should be saved with .js

extension.

Datatypes:

Datatypes specify the type of the data a

variable can hold.

Basic data types or primitive data types that are

available in JS are:

Number

String

Boolean

Null

Undefined

Both integers(numbers without decimal point)

and floating point numbers(numbers with decimal

point) are of number data type.

We can check the type of a variable by using

**typeof()**.

NaN(Not a Number):

NaN is a numeric value that represents

something that is not a number.

Ex:0/0,1+NaN

Dividing a number by 0 will lead to the value

Infinity.

VARIABLE:

A variable is a container that basically holds a

value.

Value can be a number,boolean,string,etc..

In javascript we can create a variable by using

var or let keyword.

Ex:var name=”Raju”,let name=”Raju”

console.log() is a function that is used to print

the output on to the console.

document.write() is a function that is used to

print the output on to the webpage.

RULES FOR CREATING VARIABLES:

 Variable names cannot contain spaces.

 Variable names must begin with a letter, an

underscore (\_) or a dollar sign ($).

 Variable names can only contain letters,

numbers, underscores, or dollar signs.

 Variable names are case-sensitive.

CONSTANT:

constant is a special variable that holds the

value which shouldn’t be changed.

Ex:const name=”Raju”

Boolean:

They have the values either true or false.

Booleans are basically used with conditional

statements.

Strings:

Strings are a group of characters or a collection

of characters enclosed within single or double

quotes.

we can find the length of a string by using the

property length.

we can access individual characters from the

string by using indexing concept.

Syntax:variablename[indexno]

String methods without passing arguments:

toLowerCase()=converts the string to lower

case.

toUpperCase()=converts the string in to upper

case.

trim()=removes the spaces present on both LHS

and on RHS of the string.

trimStart()=Removes the space on the LHS of

the string.

trimEnd()=Removes the space on the RHS of the

string.

toString()=It is used to convert an integer,float

number,boolean in to a string object.

String methods by passing arguments:

substring is nothing but a part of the string.

indexOf(substring)=It will check whether the

substring is present in the main string or not.if

yes,it will return the first occurence index value or

else gives the output as -1.

lastIndexOf(substring)= It will check whether

the substring is present in the main string or not.if

yes,it will return the last occurence index value or

else gives the output as -1.

charAt(indexno)=It returns the character

present at a particular index position.

charCodeAt(indexno)=It returns the ASCII value

of the character present at that index position.

concat(str2)=It is used to concatenate or

combine 2 or more strings.

syntax: str1.concat(str2,str3……)

starsWith(substring)=It is used to check

whether the string is starting with the

corresponding substring or not.

endsWith(substring)= It is used to check

whether the string is ending with the

corresponding substring or not.

fromCharCode(ASCII number)=Depending on

the ASCII number that is passed,we will get the

corresponding character.We have to use a inbuilt

class String(String object).

includes(substring)=It is used to check whether

the substring is present anywhere in the main

string or not.

repeat(count)=It is used to repeat the string as

many no of times depending upon the count value

specified.

replace(old substring,new substring)=It is used

to replace the old substring with the new

substring.

search(sub string)=It is used to check whether a

sub string is present in the main string or not.If

yes,it returns the corresponding index value or

else it returns -1 as the output.It is exactly same as

indexOf method.

slice(start value,stop value)=It is used to fetch a

group of characters present in a string.

split(sep,limit)=It is used to split the entire

string and stores the output in an array.

limit will limit the number of elements presents

in an array.

STRING TEMPLATE LITERALS:

Template literals are strings that allow

embedded expressions,which will be evaluated

and then turned in to a resulting string.

Ex:

name=”Raju”

console.log(`The name is ${name}`)

Null Data Type:

It is the value that can be given to a variable,if

that variable does not hold any value initially or at

the later stage.

Undefined Data Type:

Variables that don’t have an assigned value are

undefined.

parseInt() and parseFloat()

These 2 functions are used to parse or convert

string in to numbers.

parseInt(“24”)=24

parseInt(“24.98”)=24

parseInt(“Javascript”)=NaN

parseInt(“24Javascript”)=24

parseFloat(“24.98”)=24.98

parseFloat(“7”)=7

parseFloat(“Javascript”)=NaN

parseFloat(“24Javascript”)=NaN

OPERATORS IN JAVASCRIPT:

operators are used to perform operations

between operands.

operand is nothing but a value.

The various operators that are available in

javascript are:

ARITHMETIC OPERATORS:

These operators are used to perform arithmetic

operations between various operands.

The various Arithmetic operators are:

+,-,\*,%,/,\*\*

ASSIGNMENT OPERATORS:3

These operators are basically used to assign the

value to a variable.

The various Assignment Operators are:

=,+=,-=,%=,/=,\*=,\*\*=

COMPARISON OPERATORS:

These operators are used to perform

comparison between operands.

 The various Comparison Operators are:

==(values will be compared,but not the

datatypes),===(values will be compared,with

datatypes consideration),<,>,<=,>=,!=

BITWISE OPERATORS:

These operators are used to perform operations

on the Binary Numbers.

 The various Bitwise Operators are:

&(Bitwise AND),|(Bitwise OR),^(Bitwise XOR)

LOGICAL OPERATORS:

These operators are used to evaluate single or

multiple conditions.

The various Logical Operators are:

Logical AND(&&),Logical OR(||) and Logical NOT(!)

TERNARY OPERATOR:

It is basically used to reduce the length of the

code by evaluating a condition(if and else) in a

single line itself.

syntax:

variable=condition ? expIfTrue : expIfFalse

CONDITIONAL STATEMENTS:

Conditional statements are basically used to

execute a set of statements based on some

condition.

The various conditional statements are:

if statement

if-else-if-else statement

if-else statement

IF STATEMENT

 Statements present will get executed iff the

condition is true.

syntax:

if(condition)

{

-------------(body)

}

IF ELSE IF STATEMENT

This conditional statement is used to evaluate

multiple conditions.

Syntax:

If(condition)

{

}

else if (condition)

{

}

else

{

}

IF ELSE (or) IF ELSE IF ELSE STATEMENT

syntax:

if(condition)

{

}

else

{

}

Switch Statement:

A Modular way of writing if-else if statements.

Syntax:

Switch(case)

{

case name:

-------------

}

Type Conversion and Coercion

Type Conversion(Explicit Type Casting)-where the

programmers manually convert one data type to

another data type.

Type Coercion(Implicit Type Casting)-where

javascript automatically converts one data type to

another data type.

TYPE CONVERSION:

Number()-converts string,booleans to number.

String()-converts numbers,Booleans to string.

Boolean()-converts numbers,strings to boolean.

TYPE COERCION:

EX:console.log(“23”+30)

+ used between a string and number,the number

will be automatically converted to string.If not we

should have told String(30).

EX:console.log(“23”-30)

- used between a string and a number,string

will be converted to number.

EX:console.log(“23”\*”3”)

\* between 2 strings or between a integer and a

string or between 2 integers is the same.

/ if used between a string and a number,string will

be converted to number,only if that string

contains numbers or digits.

true/4=1/4=0.25

NOTE:When we use + operator than our integer

will be converted to string otherwise use any

other operators string will be converted to integer.

MATH OBJECT:

It contains some properties and methods to

perform mathematical operations.

It is like a module in python.

List of few properties and functions:

Math.PI

Math.round(4.9)

Math.abs(-4.5)

Math.pow(2,5)

Math.floor(6.78)

Math.ceil(6.79)

How can we generate a random numbers using

Math Object?

Math.random()

It gives us a random decimal value (or) float

value between 0 and 1.

-------------------------------------------------------------------

ARRAYS:

Arrays are data types which stores both

homogeneous as well as heterogenous type of

data.

Ex:storing the comments of a particular post,

Storing the songs in a playlist.

Arrays are represented using []

Arrays are mutable(changeable).

we can find the length of the array by using

length property.

we can access single element from the array

using index concept and group of elements using

the slice method.

we can modify the value of the element present

in the array using the index value.

ARRAY METHODS:

push()=It is used to add an element at the end

of an array.

pop()=It is used to remove an element from the

end of an array.

shift()=It is used to remove element from the

start of an array.

unshift()=It is used to add an element at the

beginning of an array.

conact()=It is used to merge the elements

present in two different arrays.

includes()=Similar to string method.

indexOf()=Similar to string method.

join()=It creates a string from an array.

syntax:array.join(sep)

splice()=It is used to remove or replace

elements.

syntax:splice(start index,delete count,items)

reverse()=It reverses the elements present in an

array.

sort()=It sorts the element present in an

array.COMPLETEY DIFFERENT!!!

**Reference types for numbers,string is completely**

**different when compared with arrays and**

**objects.**

Q)How const can we used with Arrays???

Values in array can change until the reference

remains the same!!!(length remains same).

Nested Array:

An Array inside another array.

OBJECTS:

Data is stored in the form of key-value pairs.

Represented using { } brackets.

Ex:to store the details of fitness tracker.

Ex:

const data={

“totalSteps”:4000,

“calories burnt”:300cal

}

We can access the values or update the values

by using the bracket syntax or using the key name

as property using . notation.

**We can have Nested Arrays and Nested Objects**

**present together!!!**

**How Equality operators(== and ===) works**

**between Arrays and Objects?**

LOOPS:

If we have to run certain things again and again

then we have to use loops.

Ex:repeat the logic continuously again and again

until there’s a winner or drawn,sum all numbers in

an array,display all the facebook comments by

iterating over an array.

Types of loops in javascript:

1)for loop

2)for..of..loop

3)for…in….loop

4)while loop

1)for loop

Syntax:

For(initialexpression;condition;inc/dec)

{

body

}

**How Infinite loops works?**

**Working of for loop + Arrays**

**Nested For Loops**

While loop:

While loops will get executed as long as the

condition is true.

Transfer Statements:

Break and Continue

These Statements are used to transfer the

control the control from one part of the program

to some other part.

for-of loop:

It is an easy way of iterating over an

Array,strings.

It is not supported on Internet Explorer!!!

syntax:

for(variable of sequence)

{

Statement

}

**Can we use for of loop on OBJECTS:**

Object.keys(object)displays only the keys.

Object.values(object)displays only the values.

Object.entries(object)displays keys and values

together in array format.

for-in loop:

Loops over the keys in an object

Syntax:

for(variable in object)

{

statement

}

We can use for-in loop to loop over

Strings,Arrays and Objects.

**Use of Strict Mode??**

It creates visible errors in the developers

console and also helps developers if reserved

words are used as variable names.

It should be always the first line in javascript

file.

we can activate it by using ‘use strict’.

FUNTIONS IN JAVASCRIPT:

Functions allows us to write reusable,modular

code and it can be called any time!!!

syntax:

function functionName()

{

//do something

}

**Create a function that gives you all the dice**

**numbers!!!**

A function can accept parameters and we can

pass arguments as input values.

No of Parameters and No of Arguments should

always be **EQUAL**.

A function in javascript can return only one

value at a time.

return statement indicates the end of a function

body.

To return multiple values from a function, you

can pack the return values as elements of

an array or as properties of an object.

Ex:

function **getNames**() {

let fname="Raju"

let lname="Kumar"

return [fname, lname];

}

data=**getNames**()

console.**log**(data)

function **getNames**() {

let fname="Raju"

let lname="Kumar"

return {

fName:fname,

lName:lname

};

}

data=**getNames**()

console.**log**(data)

we can pass arguments as keyword

arguments,we can define a default argument in

case user forgets a value.

A single parameter can accept any number of

arguments by using …

------------------------------------------------------------------

PRACTICE-1:

i. Write a isValidPassword function

ii. It accepts 2 arguments: password and

username

iii. Password must:

be at least 8 characters

cannot contain spaces

cannot contain the username

iv. If all requirements are met, return

true,otherwise false.

PRACTICE-2:

i. Write a function to find the average value in

an array of numbers.

-----------------------------------------------------------------

let data=[10,20,30,40]

let total=0

function calculateAverage()

{

for(let i of data)

{

total=total+i

}

let count=data.length

console.log("The Average value is"+' '+`$

{total/count}`)

}

calculateAverage()

PRACTICE-3:

i. Write a function called isPangram, which

checks to see if a given sentence contains

every letter of the alphabet. Make sure you

ignore string casing!

ii. A pangram is a sentence that contains every

letter of the alphabet, like:

**"The quick brown fox jumps over the lazy dog"**

function isPanagram(data)//data="learning javascript t

akes time"

{

let characters="abcdefghijklmnopqrstuvwxyz"

for(let i of characters)//i="a"

{

if(data.indexOf(i) == -1)

{

console.log("Not a Panagram")

}

else

{

console.log("It is a Panagram")

}

}

}

isPanagram("the quick brown fox jumps over a lazy dog"

)

**Function Scope:**

Scope is not but variable visiblility.

The location where the variable is defined

dictates where we have access to that variable.

Variables created inside the function are scoped

to that function.

Variables created outside the function can be

accessed inside the function as well as outside the

function.

In this scope all the three let,var and const

behave in the same way.

**Block Scope:**

A **block scope** is the area within if, switch

conditions or for and while loops not with

functions.

Generally speaking, whenever you see {curly

brackets}, it is a **block**

let and const have a Block Scope,i.e Variables

declared inside a { } block cannot be accessed

from outside the block,whereas its completely

different in case of var.

Redeclaring a variable within a block using let

and const is completely different w.r.t var.

var x = 10;

// Here x is 10

{

var x = 2;

// Here x is 2

}

// Here x is 2

let x = 10;

// Here x is 10

{

let x = 2;

// Here x is 2

}

// Here x is 10

const x = 10;

// Here x is 10

{

const x = 2;

// Here x is 2

}

// Here x is 10

**Lexical Scope:**

A variable created inside the outer function will

be available to the inner function.

A variable created inside the inner function will

not be available to the outside function.

Function Declarations vs Function Expressions:

Function Declarations:

until now we have created the functions using

function keyword.

Function Expressions(Anonymous Function)

Functions without a name and assigning the

function basically to a variable.

Variable created will hold the function value.

The Main difference between function

declaration and expression is,we can the fn before

its defined in case of function declaration,where

as its not possible in function expression.This

happens because of **Hoisting.**

ARROW FUNCTIONS:

It is a shorter form of Function Expression.

{} brackets can be replaced with () and almost

nothing if there is a single line inside the function.

Here as well,we cannot call a function before

creating it!!!

Higher Order Functions:

Functions that operate on/with other functions

are called as **Higher Order Functions**.

**Function accepting another Function as**

**Argument.**

**A Function can return another function**.

function outer()

{

return function()

{

return function()

{

console.log("Welcome")

}

}

}

let result=outer()

let result1=result()

result1()

**NOTE**: The function that is normally passed as an

argument are called as **Callback Functions.**

**FOR EACH Function:**

Accepts a callback function and calls the

function once per element in the array.

**syntax:**

Array.forEach(function(element,index,array))

Object.forEach(function(element,index,array))

 It can used on Arrays and Objects present

within Array.

**ForEach on ARRAY:**

const data=[1,2,3,4]

const result=data.forEach((n) =>

{

return console.log(n\*n)

})

console.log(result)

**ForEach on Array of Objects:**

const data=[

{

"name":"Raju",

"age":34,

"marks":50

},

{

"name":"John",

"age":23,

"marks":70

},

]

const result=data.forEach((element) =>

{

return element.name.toUpperCase()

})

console.log(result)

**Map function:**

It creates a new array with the results of calling

a callback .

**syntax:**

Array.map(function(element,index,array))

Object.map(function(element,index,array))

**Ex:** creating a new array by combining both first name

and last name.

const persons = [

{firstname : "Malcom", lastname: "Reynolds"},

{firstname : "Kaylee", lastname: "Frye"},

];

const data=persons.map((a)=>

{

return [a.firstname,a.lastname].join(" ")

})

console.log(data)

**Filter Function:**

It creates a new array with all the elements that

pass the test implemented by the provided

function.

syntax:

Array.filter(function(element,index,array))

Object.filter(function(element,index,array))

**FILTER ON ARRAYS:**



const words = ['spray', 'limit', 'elite', 'exuberant',

'destruction', 'present'];

const result=words.filter((word) =>

{

return word.length > 6

})

console.log(result)

**FILTER ON ARRAY OF OBJECTS:**

const data=[

{

"name":"Raju",

"age":34

},

{

"name":"John",

"age":23

},

]

const result=data.filter((data) =>

{

return data.name.startsWith("R")

})

console.log(result)

**Reduce Function:**

It executes a reduce function on each element

of the array,resulting in a single value.

A callback passed to a reduce function should

accept a minimum of 2 parameters.Normally we

call them as accumulator and currentvalue.

accumulator stores the end value of every

reduce.

current value refers to each element in a array.

In beginning accumulator stores the first value

of an array,currentvalue stores the second value.

TO FIND MAXIMUM ELEMENT IN ARRAY:

const array1 = [1, 2,45, 3, 4];

const result=array1.reduce((accumulator, currentvalue)

=>

{

if(accumulator > currentvalue)

{

return accumulator

}

else

{

return currentvalue

}

})

console.log(result)

we can also pass a second parameter to reduce

where we can provide initial value for the

accumulator.

**Sum of values in an object array using Reduce:**

let initialValue = 0

let sum = [{x: 1}, {x: 2}, {x: 3}].reduce(

(accumulator, currentValue) => accumulator + curre

ntValue.x

, initialValue

)

console.log(sum)

**Counting instances of values in an object using**

**Reduce:**

const data=[

{

"name":"Raju",

"age":34,

"marks":98

},

{

"name":"John",

"age":23,

"marks":76

},

]

//groupedBooks={}

const result=data.reduce((groupedBooks, book)=>

{

const key=book.marks//key=98

if(!groupedBooks[key])

{

groupedBooks[key]=[]

groupedBooks[key].push(book)

return groupedBooks

}

},{})

console.log(result)

**FIND method on ARRAY!!!**

It returns the value of the first element in the

array that satisfies the provided testing function.

let movies=[

"Mr.Ramu",

"mr.Steve",

"Mr.John"

]

const result=movies.find((name) =>

{

return name.includes("Mr")

})

console.log(result)//Mr.Ramu

**Find Method on Array of Objects:**

const data=[

{

"name":"Raju",

"age":34,

"marks":50

},

{

"name":"John",

"age":23,

"marks":70

},

]

const result=data.find((mark) =>

{

const myMark=mark.marks

if(myMark > 35)

{

return myMark

}

})

console.log(result)

**Every Function:**

It checks whether all the elements in the array

pass the provided condition or not.It returns a

Boolean Value.

syntax:

Array.every(function,index,array)

const word=["Raju","Rani","John"]

const result=word.every((word)=>

{

return word.length == 4

})

console.log(result)

Some Function:

It is similar to every function,but returns if any

of the array elements pass the call back function.

syntax:

Array.some(function,index,array)

const array = [1, 2, 3, 4, 5];

// checks whether an element is even

const even = (element) => element % 2 === 0;

console.log(array.some(even));

// expected output: true

**Sort Function:**

It sorts the elements based on the ascending

order of the string.

Array.sort(compareFunc(a,b))

If compareFunc(a,b) returns less than 0,sort a

before b.

 If compareFunc(a,b) returns greater than 0,sort

b before a.

 If compareFunc(a,b) returns 0,leave a and b

unchanged w.r.t each other.

const data=[10,3,20,13,40,50]

const result=data.sort((a,b)=>

{

return b-a

})

console.log(result)//descending order

**Spread Operator:**

Spread syntax (...) allows an iterable such as an

array expression or string to be expanded in places

where zero or more arguments (for function calls)

or elements (for array literals) are expected, or an

object expression to be expanded in places where

zero or more key-value pairs (for object literals)

are expected.

**How Spread operator is used in Functions?**

function collect(a,b,c)

{

return a+b+c

}

let data=[10,20,30,40]

console.log(collect(...data))//(10,20,30,40)

It expands an iterable(array,string,etc..)in to a

list of arguments.

**How Spread operator is used on Arrays?**

Create a new array using an existing array.

It basically spreads the elements from one array

into a new array.

const a=[10,20,30]

const b=[40,50,60]

const c=[70,80,90]

const e=[...a,...b]

console.log(e)

We can apply this operator for every element in

an array,where every character gets splitted.

console.log([..."Ramu",..."John"])

**How Spread operator is used on Objects?**

let obj1 = { foo: 'bar', x: 42 };

let obj2 = { foo: 'baz', y: 13 };

let clonedObj = { ...obj1 };

// Object { foo: "bar", x: 42 }

let mergedObj = { ...obj1, ...obj2 };

// Object { foo: "baz", x: 42, y: 13 }

Values can also be overwritten!!!

let obj1 = { foo: 'bar', x: 42 };

let obj2 = { foo: 'baz', y: 13 };

let data={

...obj1,

x: 45

}

console.log(data)//{ foo: ‘bar’, x: 45 }

**Rest Operator:**

It looks like spread,but its not!!!

It allows a function to accept indefinite number

of arguments as an array.

The rest parameter should always be the last

parameter in the function definition.

**Destructuring:**

It is a clean syntax to unpack values from

arrays,properties from objects in to distinct

variables.

Array Destructuring:

The process of unpacking the values from an array and

storing it in variables in a compact way

is only called as Array Desctructing

const data=["blue","green","orange","Ramu","Raju"]

const [color1,color2,color3]=data

const data=["blue","green","orange","Ramu","Raju"]

const [color1, , , color3]=data

const data=["blue","Ramu","Raju"]

const [colors, ...names]=data

**Objects Destructuring:**

Here we unpack based on the key name.

let data={

name:"Ramu",

age:27,

country:"India"

}

const {name, age, ...others} = data

Nested Destructuring:

const persons = [

{

firstname : "Ramu",

"age":34

},

{

firstname : "Simi",

"age":20

},

]

const [ , {age}] = persons

Destructuring Params:

const person=

{

firstname : "Simi",

age:20

}

function print(data)

{

const {firstname, age} = data

console.log(firstname)

console.log(age)

}

print(person)

Simplest way is:

const person=

{

firstname : "Simi",

age:20

}

function print({firstname,age})

{

console.log(firstname)

console.log(age)

print(person)

The above concept can also be implemented for

Arrays.

**Shorthand Object Properties**:

function calculate(collect)

{

const max=Math.max(...data)

const min=Math.max(...data)

const add=collect.reduce((accumulator,currentvalue

)=>accumulator+currentvalue)

const avg=add/collect.length

return{

max,

min,

add,

avg

}

}

let data=[10,20,30,40]

const stats=calculate(data)

console.log(stats)

**Adding methods to Objects:**

const math={

"add":(x,y)=>x+y,

"sub":(x,y)=>x-y,

"mul":(x,y)=>x\*y,

"div":(x,y)=>x/y,

}

console.log(math.add(4,5))

console.log(math.sub(4,5))

console.log(math.mul(4,5))

console.log(math.div(4,5))

**this keyword(!!!IMPORTANT)**

It refers to the object on which we call our

function.

window is the default object in the browser.

function wish()

{

console.log("Hi")

console.log(this)

}

Wish()

window.wish()

let and const are not added to the global scope

of the browser i.e window.

const person={

first:"Simi",

last:"Singh",

nickName:"Sim",

fullName()

{

console.log(first)//not possible

console.log(this.name)//possible

}

}

Here,the **this** is not holding the window

object,instead its holding the person object.

Now we can access the properties of that object

using this keyword.

**DOM(Document Object Model)**

It links JS object to HTML file.

It is used to add some interactive nature to the

HTML elements present on the webpage.

Whenever we execute a webpage,browser

reads it an represents the HTML elements+CSS if

any in to corresponding Javascript Objects.

we can see it by using console.dir(document)

HTML to DOM converter:

https://software.hixie.ch/utilities/js/live-domviewer/#

Selecting the Elements:

SelectManipulate

getElementById()

getElementsByTagName()

getElementsByClassName()

querySelector()All in one

querySelectorAll()

**innerText and textContent**used for accessing

data.

innerText gives the text between the tags.

Ex;text present between heading tags,within ul

tag(with li tag),text within complete body.we can

update the inner text as well.

textContent works mostly on a paragraph.

**innerHTML** gives text as well as the tags inside

a given element.using innerHTML we can create

an element with a tag associated with it.

Ex:form.innerHTML=”<b>Hi</b>”

value property gets and sets the values

associated with an element,may be with a input

field present inside a form

we can use checked property for check boxes.

**getAttribute() and setAttribute() methods**

getAttribute() returns the value of a specified

attribute on the element.If the given attribute

does not exist,the value returned will be null or “”

setAttribute()=sets the value to a particular

attribute present with an element.

changing the src value of an image,href of a

anchor tag,etc..

**syntax:**

Element.setAttribute(attributename,attributevalu

e)

we can access src attribute value associated

with image tag and href attribute associated with

anchor tag.

Finding Parent/Children and Sibblings:

parentElement property

children property

nextElementSibbling property

previousElementSibbling property.

Altering Styles:

Ex:Element.style.property

Here the property names are in camel case.

let alll=document.querySelectorAll("li")

let colorss = ["red","orange","yellow"]

alll.forEach((el,i)=>

{

let color = colorss[i]

el.style.color=color

})

getComputedStyle() gives list of all the style

properties on a particular element.

Manipulating Classes:

let todo=document.querySelector(".todo")

todo.style.color="red";

todo.style.textDecoration="line-through"

todo.style.opacity="50%"

Instead of giving the style to a element one by

one.we can define all the styles in a CSS file and

link that to our JS file using setAttribute method.

todo.setAttribute("class","done")

But the disadvantage is original css style will be

lost,so we have to use classList(not exactly an

array).

we have quite a few number of methods on

classList,which are very much helpful.

CREATING ELEMENTS:

we can use create element.

Adding Elements to the webpage:

appendChild(by default it’s the last element)

insertBefore(newelement,oldelement)

Ex:

p.insertAdjacentElement('beforebegin',i)

beforebegin,afterbegin,beforeend,afterend are

values to insertAdjancentElement

append()=used to add multiple elements at a

time at the end.

prepend()=used to add multiple elements at a

time at the end.

removeChild()=parentelement.removeChild(chil

delelement)

remove()=element.remove()

DOM Events:

It is a mechanism of responding to user inputs

and actions.

onclick property on a button,as well as on

heading tags.

ondblclick()

onmouseenter()

addEventListener()

{once:true}

1. What are the modules in javascript?

2. How to validate email in javascript?

3. Explain this keyword in javascript?

4. Explain arrow function in javascript?

5. What are the asynchronous functions?

6. What is a callback in javascript?

7. What is the use of callbacks in javascript?

8. What are the window methods?

9. Explain the properties of window objects?

10. What are variables and functions in BOM?

11. What is validation?

12. How to find the html elements in DOM?

13. Is the html dom can be accessed with the

javascript?

14. Which symbol is used for comments in

javascript?

15. What would be the result of 3+2+”7”?

16. Is it possible to break the javascript code in a

single line?

17. What are the global variables?

18. What do you mean by null in javascript?

19. Is javascript case sensitive?

20. What is object and class in javascript?

21. What is a script ?

22. What is the use of script in javascript?

23. What are the variables in javascript?

24. What are the rules to declare the variables in

javascript ?

25. What are the datatypes in javascript?

26. What is the difference between let , var, and

const?

27. What are the operators in javascript?

28. What are the unary and binary operators?

29. What are the types of operators in javascript?

30. Explain arithmetic operator?

31. What is the use of a concatenation operator?

32. Explain relational operators in javascript?

33. Explain logical operators in javascript?

34. Explain increment and decrement operators in

javascript?

35. What are the comments in javascript?

36. What are commonly used built-in functions in

javascript?

37. Explain the statements in javascript?

38. Explain function in javascript?

39. What is event handling in javascript?

40. Explain the mouse events in javascript?

41. Explain the keyboard events in javascript?

42. Explain the form events in javascript?

43. Explain js object?

44. Explain a built-in object?

45. Explain string object?

46. Explain a math object?

47. Explain a date object?

48. Explain a number object?

49. Explain an array object?

50. How to write the array in javascript?

51. What is meant by html dom properties?

52. What is meant by dom methods?

53. Why is javascript a dynamically typed

language?

54. Tell the scope of the var , let and the const?

55. Create a variable of type string and try to add a

number in it?

56. Use typeof operator to find the datatype of the

string in the last question?

57. Create a const object in javascript can you

change it to hold a number later?

58. Try to add a new key to the const object in

problem 57 were you able to do it?

59. Write a javascript program to create a word

meaning dictionary of 5 words?

60. How many data types are present in javascript?

**Javascript Interview Questions**

1. **What is JavaScript?**

Answer: JavaScript is a high-level, interpreted programming language that is primarily used to create interactive and dynamic content on web pages. It is an essential part of web development, alongside HTML and CSS.

1. **What are the different data types in JavaScript?**

Answer: JavaScript supports several data types, including:

Primitive types: string, number, boolean, null, undefined, symbol, and bigint

Objects: Object, Array, Function, Date, RegExp, and more.

1. **What is the difference between let, const, and var?**

Answer:

var: Function-scoped or globally-scoped if declared outside a function, can be re-declared and updated.

let: Block-scoped, can be updated but not re-declared within the same scope.

const: Block-scoped, cannot be updated or re-declared. However, the properties of objects declared with const can be modified.

1. **What is a closure in JavaScript?**

Answer: A closure is a function that retains access to its lexical scope, even when the function is executed outside that scope. This allows a function to "remember" the environment in which it was created.

1. **Explain event delegation in JavaScript.**

Answer: Event delegation is a technique where a single event listener is added to a parent element to manage events for its child elements. This is efficient as it reduces the number of event listeners and utilizes event bubbling.

1. **What is the difference between == and === in JavaScript?**

Answer: == (loose equality) checks for equality after performing type conversion if necessary, while === (strict equality) checks for equality without type conversion, meaning both value and type must be the same.

1. **What is the purpose of the this keyword in JavaScript?**

Answer: The this keyword refers to the object it belongs to. Its value depends on the context in which it is used, such as within a method, a function, or in global scope.

1. **What are promises in JavaScript?**

Answer: Promises are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. A promise can be in one of three states: pending, fulfilled, or rejected.

1. **What is an arrow function and how is it different from a regular function?**

Answer: Arrow functions provide a shorter syntax for writing functions and do not have their own this binding. Instead, this is lexically inherited from the surrounding scope.

// Regular function

function add(a, b) {

return a + b;

}

// Arrow function

const add = (a, b) => a + b;

1. **What are the different ways to create an object in JavaScript?**

Answer: Objects can be created in several ways:

Using object literals:

const obj = { key: 'value' };

Using the new keyword with a constructor function:

function Person(name) {

this.name = name;

}

const person1 = new Person('John');

Using Object.create() method:

const proto = { key: 'value' };

const obj = Object.create(proto);

1. **What is asynchronous programming and how does JavaScript handle it?**

Answer: Asynchronous programming allows the program to perform tasks without blocking the main thread. JavaScript handles asynchronous operations using callbacks, promises, and async/await syntax.

1. **What is the event loop in JavaScript?**

Answer: The event loop is a mechanism that allows JavaScript to perform non-blocking I/O operations, despite being single-threaded. It continuously checks the call stack and the task queue to execute code, collect events, and execute queued tasks.

1. **What is the difference between call, apply, and bind?**

Answer:

call: Invokes a function with a specified this value and arguments provided individually.

apply: Invokes a function with a specified this value and arguments provided as an array.

bind: Creates a new function that, when called, has its this value set to a specified value, with a given sequence of arguments preceding any provided when the new function is called.

function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person = { name: 'Alice' };

greet.call(person, 'Hello', '!'); // Hello, Alice!

greet.apply(person, ['Hi', '.']); // Hi, Alice.

const greetPerson = greet.bind(person, 'Hey');

greetPerson('?'); // Hey, Alice?

1. **Explain hoisting in JavaScript.**

Answer: Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compilation phase. This means variables declared with var and functions can be used before their actual declaration in the code.

1. **What is the difference between synchronous and asynchronous functions**?

Answer: Synchronous functions are executed sequentially, blocking the execution of subsequent code until they finish. Asynchronous functions allow the program to continue executing subsequent code while they complete in the background, typically using callbacks, promises, or async/await for handling the result.

1. **What is event loop? What is precedence of event loop?**

The event loop is a fundamental concept in JavaScript that allows for asynchronous programming by managing

the execution of code, collecting and processing events, and executing queued sub-tasks. It&#39;s a part of the

JavaScript runtime environment and is crucial for handling operations that might take some time to complete,

such as I/O operations, without blocking the main thread.

1. **How the Event Loop Works**

Call Stack: This is where the JavaScript engine keeps track of function calls. When a function is invoked, it&#39;s added

to the call stack. When the function completes, it&#39;s removed from the stack.

Web APIs: These are browser-provided APIs that handle operations like setTimeout, DOM events, HTTP requests

(via XMLHttpRequest or Fetch), and more. When these operations are completed, they send their callbacks to

the task queue.

Task Queue (or Callback Queue): This queue contains callbacks for events like user interactions, timers, or

network responses. These tasks are processed after the current execution context (call stack) is empty.

Microtask Queue: This is a special queue for microtasks, which include promises and MutationObserver

callbacks. Microtasks are given higher priority over tasks in the task queue.

Event Loop: The event loop continuously checks the call stack to see if it&#39;s empty. If the call stack is empty, it

looks at the microtask queue first and processes all microtasks before moving to the task queue. This ensures

that promises and other microtasks are handled as soon as possible.

1. **Among setTimeout() and promise() which one will take precedence?**

In JavaScript, when comparing setTimeout() and Promise (or more specifically, the .then() or .catch() methods of

a promise), promises take precedence over setTimeout() due to the way the event loop handles microtasks and macrotasks.

Explanation

Promises: When a promise resolves, its .then() and .catch() handlers are placed in the microtask queue.

Microtasks are given higher priority and are executed before any tasks in the macrotask queue.

setTimeout(): The callback function passed to setTimeout() is placed in the macrotask queue (also known as the

task queue). Tasks in this queue are processed only after all the microtasks have been completed.

Event Loop Order of Operations

Execute all code in the current call stack.

Process all the tasks in the microtask queue.

Process the next task from the macrotask queue.

Example

Consider the following code snippet to demonstrate the precedence:

console.log(&#39;Start&#39;);

setTimeout(() =&gt; {

console.log(&#39;setTimeout&#39;);

}, 0);

Promise.resolve().then(() =&gt; {

console.log(&#39;Promise&#39;);

});

console.log(&#39;End&#39;);

Execution Flow:

Call Stack: Execute console.log(&#39;Start&#39;) which logs &quot;Start&quot;.

Call Stack: Execute setTimeout which schedules the callback in the macrotask queue.

Call Stack: Execute Promise.resolve().then(...) which schedules the .then() callback in the microtask queue.

Call Stack: Execute console.log(&#39;End&#39;) which logs &quot;End&quot;.

After the call stack is empty:

Microtasks: Process the promise&#39;s .then() callback which logs &quot;Promise&quot;.

Macrotasks: Process the setTimeout callback which logs &quot;setTimeout&quot;.

So, the output will be:

Start

End

Promise

setTimeout

Summary

Promises and their .then() handlers are placed in the microtask queue and are executed before the tasks in the

macrotask queue.

setTimeout() callbacks are placed in the macrotask queue and are executed after all microtasks have been

processed.

Therefore, promises take precedence over setTimeout() in the event loop.

1. **Where do you mostly use rest operator?**

The rest operator (...) in JavaScript is used to handle function arguments, array elements, and object properties

in a flexible way. Here are the primary scenarios where the rest operator is commonly used:

1. Function Parameters

The rest operator allows you to represent an indefinite number of arguments as an array. This is especially useful

when creating functions that accept a variable number of arguments.

Example:

function sum(...numbers) {

return numbers.reduce((total, num) =&gt; total + num, 0);

}

console.log(sum(1, 2, 3)); // 6

console.log(sum(4, 5, 6, 7)); // 22

2. Destructuring Arrays

The rest operator can be used in array destructuring to capture the remaining elements into a new array.

Example:

const [first, second, ...rest] = [1, 2, 3, 4, 5];

console.log(first); // 1

console.log(second); // 2

console.log(rest); // [3, 4, 5]

3. Destructuring Objects

Similarly, the rest operator can be used in object destructuring to collect the remaining properties into a new

object.

Example:

const { a, b, ...rest } = { a: 1, b: 2, c: 3, d: 4 };

console.log(a); // 1

console.log(b); // 2

console.log(rest); // { c: 3, d: 4 }

4. Function Arguments

When dealing with function arguments, especially in functions where you don&#39;t know how many arguments will

be passed, the rest operator can help you manage them efficiently.

Example:

function logArguments(...args) {

args.forEach(arg =&gt; console.log(arg));

}

logArguments(&#39;one&#39;, &#39;two&#39;, &#39;three&#39;);

// one

// two

// three

Summary

The rest operator is versatile and simplifies handling variable numbers of parameters and properties in functions,

arrays, and objects. Its main uses are:

Function Parameters: Capturing all arguments passed to a function.

Array Destructuring: Collecting remaining elements in an array.

Object Destructuring: Collecting remaining properties in an object.

Managing Function Arguments: Simplifying the handling of a variable number of arguments.

Using the rest operator makes code more concise and readable, especially when dealing with collections of data

or dynamic parameters.

1. **What is shallow copy and deep copy?**

In JavaScript, the concepts of shallow copy and deep copy refer to the ways in which objects or arrays are

duplicated. The key difference lies in how these copies handle nested objects or arrays.

Shallow Copy

A shallow copy of an object or array is a new object or array that has copies of the references to the original

object or array&#39;s elements. If the original object contains other objects or arrays as its properties, the shallow

copy will have references to those same objects or arrays, not duplicates.

Deep Copy

A deep copy of an object or array is a new object or array that is an entirely independent copy of the original.

This means that all levels of nested objects or arrays are also copied, resulting in a completely separate object or

array that shares no references with the original.

Methods for Creating Copies

Shallow Copy Methods:

Object.assign():

let shallowCopy = Object.assign({}, originalObject);

Spread Operator (...):

let shallowCopy = { ...originalObject };

let shallowCopyArray = [...originalArray];

Array.prototype.slice():

let shallowCopyArray = originalArray.slice();

Deep Copy Methods:

Manual Recursion:

Write a custom function to recursively copy nested objects or arrays. This method is the most flexible and can

handle special cases, but it is more complex to implement.

function deepCopy(obj) {

if (obj === null || typeof obj !== &#39;object&#39;) return obj;

if (Array.isArray(obj)) {

return obj.map(item =&gt; deepCopy(item));

}

const copy = {};

for (let key in obj) {

if (obj.hasOwnProperty(key)) {

copy[key] = deepCopy(obj[key]);

}

}

return copy;

}

JSON.parse(JSON.stringify()):

This method is simpler and works well for plain objects and arrays but has limitations. It cannot handle functions,

undefined, Infinity, NaN, and other non-serializable data.

let deepCopy = JSON.parse(JSON.stringify(originalObject));

Summary

Shallow Copy: Creates a new object or array with copies of the original&#39;s references. Changes to nested objects

or arrays will affect both the original and the copy.

Deep Copy: Creates a new object or array that is entirely independent of the original, including all nested objects

or arrays. Changes to the deep copy will not affect the original.

Understanding the difference between shallow and deep copies is crucial for managing data in JavaScript,

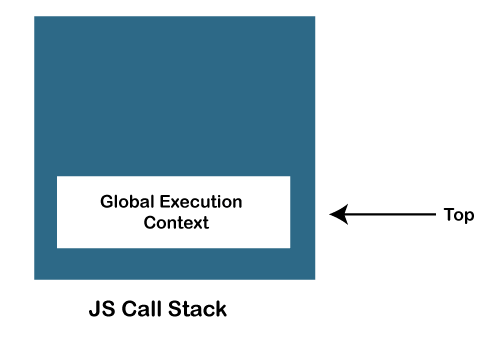
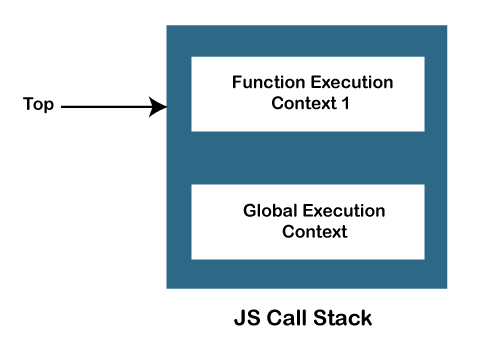
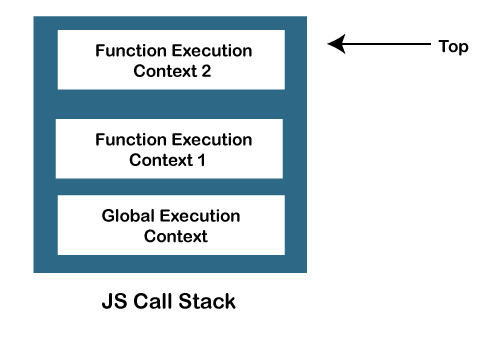
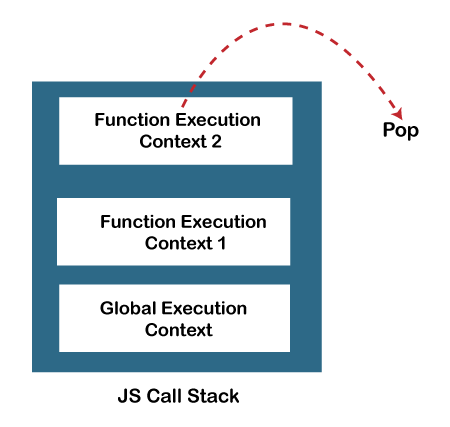
especially when dealing with complex objects and nested structures.

What is JS Call Stack

The [JavaScript](https://www.javatpoint.com/javascript-tutorial) execution contexts (Global execution context and function execution context) are executed via the JavaScript engine. In order to manage these execution contexts, the JS engine uses the call stack. So, the JS call stack is a [data structure](https://www.javatpoint.com/data-structure-tutorial) that keeps track information of the functions being called and executed. Thus, if the user invokes a function for execution, the specified function gets pushed/added in the call stack, and when the user returns from a function, it means the function is popped out from the call stack. Thus, call stack is a normal [stack data structure](https://www.javatpoint.com/data-structure-stack) that follows the stack order principal, i.e., LIFO (Last In First Out).

### **Role of JavaScript Call Stack**

There are the following points where the call stack is being used by the JS engine:

* When any script is executed by the user, the JS engine creates a Global execution context and then adds it on the call stack and at the top of the stack so that it may get executed.  
  
* When any function is invoked, the JS engine creates a Function execution context and adds it on the stack and at the top of the stack so that the invoked function may get executed.  
  
* In case a function invokes another function, the JS engine creates a Function execution context for the invoked function, adds it to the top of the stack, and begins the execution.   
  
* When any function execution gets completed, the JS engine pops it out of the stack and continues the execution of the other functions stored in the stack.  
  
* If no space is left in the stack and we try to push more functions, it throws a "stack overflow" error, and if no further execution context is present in the call stack, it throws a "Stack Underflow" error.

### **JavaScript Call Stack Example**

Let's see an example to understand the use of the JavaScript Call Stack function:

function getSum(x, y) {

return x+ y;

}

function findavg(x,y) {

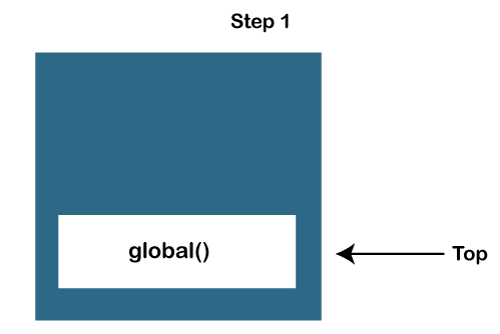
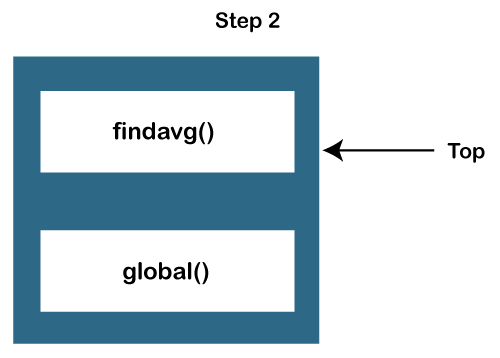
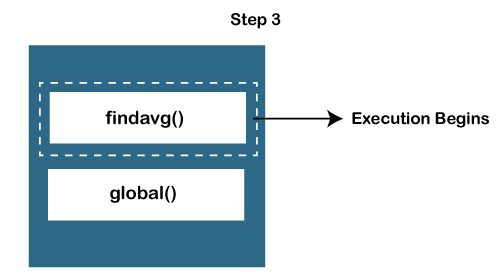
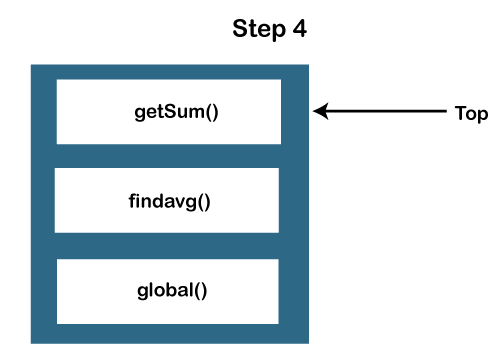
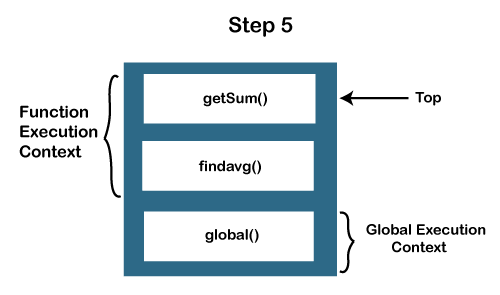
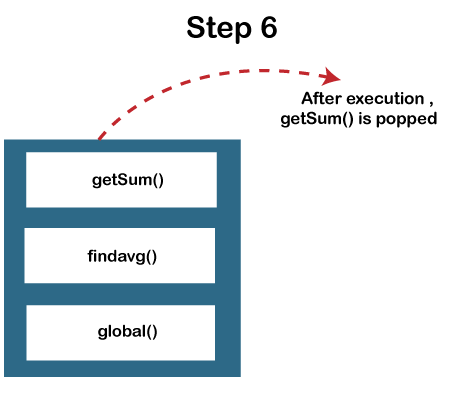
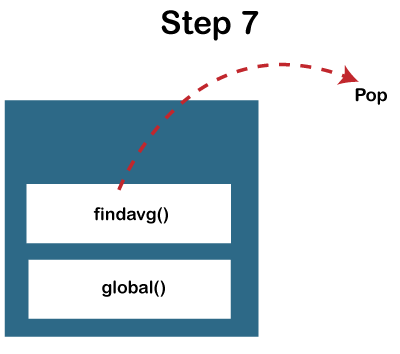
return getSum(x,y) / 2;

}

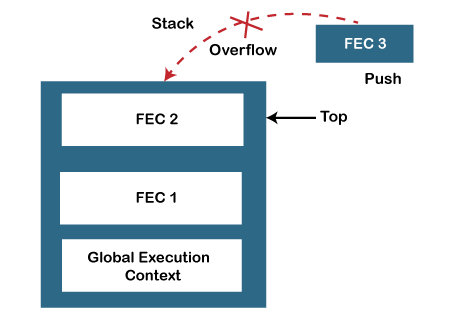
let z = findavg(10, 20);

### **How the code works**

In the above code, we have created two functions, getSum () and findavg (), and the execution of the script begins in the following described steps:

1. When the execution of the script begins, the JS engine initially creates a global execution context (i.e., global () function) and adds it to the top of the call stack.
2. The global execution moves to the execution phase of the life cycle after entering the creation phase, as you can see in the below image:  
   
3. The findavg (10, 20) function gets invoked, and so the JS engine creates the function execution context for it. Then push it on the top of the call stack.
4. So, now in the call stack, two functions are pushed, i.e., global () and findavg(), and on the top of the stack, the findavg() function is present, as you can see in the below image  
   
5. The JS engine begins the execution of the findavg () function because it exists on the top of the stack, as you can see in the image:  
   
6. As in the code, the getSum () function is invoked inside the findavg () function definition, so the JS engine creates a function execution context for the getSum () function and pushes it on the top of the stack.
7. Now, in the stack, there are three functions present, which are global (), findavg (), and getSum () functions, as you can see in the below image:  
     
   There are two functional execution contexts and a global execution context as you can see below:  
   
8. So, the JS engine executes the getSum () function first and pops it out of the call stack.  
   
9. Similarly, the findavg () function gets executed and gets out of the call stack.  
   
10. As both executions of the functions are completed, and no other function for execution is left in the call stack. The JS engine stops the execution of the call stack and moves for the other execution tasks.

### **When does call stack overflow**



The overflow condition occurs when there is no more space left in the call stack, or the condition may occur when there is a recursive function that has no exit point. The JavaScript call stack is defined with a fixed size that depends on the implementation of the host environment (which is either the Node.js or web browser). So, when the limit of the defined size of the stack is exceeded, then stack overflow occurs. Thus, it throws a stack overflow error.

**Example:**

The below example describes the stack overflow condition:

function test(){

test();

}

test();

So, in the above code, we can see that we have invoked the **test ()** function recursively, which means this function will execute until the host environment maximum call size exceeded, and thus the stack throws the stack overflow error.

**Point to be noted:**

JavaScript is a synchronous and single-threaded programming language. It means that when any script gets executed, then the JS engine executes the code line by line, starting from top to bottom. So, the JavaScript engine has only one call stack, and it can do only one thing at a time.